Version 0.6b
May 22, 2001

NIST DASE PROTOTYPE IMPLEMENTATION

PAE IMPLEMENTATION GUIDE

NOTE: This document is considered a DRAFT version and may not be correct in all aspects regarding the current NIST DASE Prototype Implementation. This document is for informational purposes only and may not be redistributed. The final, up-to-date version of this document will have a version number of 1.0 or higher.

1. INTRODUCTION

This document describes the National Institute of Standards and Technology (NIST) prototype implementation of the Digital TV Application Software Environment (DASE) Procedural Application Environment (PAE). The purpose of the implementation is to provide a programming environment for DASE applications. The implementation relies on the Hardware Abstraction Layer (HAL) for access to system data and resources that are normally system dependent. THE HAL isolates the implementation from the underlying Set-top Box (STB) environment, including transport stream data management. In the current NIST implementation, the HAL communicates with the NIST Simulation for STB functionality and transport stream data access.

The NIST DASE Prototype Implementation PAE Implementation Guide provides the reader with a behind the scenes view of the implementation internals. It is hoped that this document will provide insight to other developers in their implementations and help to clarity the interpretation of Application Programming Interfaces (API) specified in the DASE PAE document [A100/2].

2. DEFINITION OF TERMS

2.1 Acronyms and Abbreviations

API Application Programming Interface
ATSC Advanced Television Systems Committee
AWT Abstract Window Toolkit
DASE Digital TV Application Software Environment
DAVIC Digital Audio Visual Council
DET Data Event Table
DST Data Service Table
EIT Event Information Table
ETT Extended Text Table
HAL Hardware Abstraction Layer
HAVi Home Audio Video Interoperability
JDK Java Development Kit
JMF Java Media Framework
JNI Java Native Interface
JRE Java Runtime Environment
JVM Java Virtual Machine
MGT Master Guide Table
NIST National Institute of Standards and Technology
PAE Procedural Application Environment
PCR Program Clock Reference
PSIP Program and System Information Protocol
P-Java Personal Java
RI Reference Implementation
RRT Region Rating Table
SDF Service Description Framework
STB Set-top Box
STT System Time Table
VCT Virtual Channel Table
UML Unified Modeling Language
URI Uniform Resource Identifier

2.2 Terms

DASE application:

data broadcast application:

data carousel:

data-enhanced A/V channel: corresponds to a television service that also carries data components.

data service:

hardware abstraction layer:

stand-alone data channel: corresponds to virtual channels with only data components.

service:

television service: corresponds to virtual channels with audio/video components.

xlet:

3. API IMPLEMENTATION MAPPINGS AND STATUS

3.1 Status Table Guidelines

Throughout this document the completion status of the implementation at various levels is given. Table 1 provides descriptions for the stages of completion. This will give the reader some indication of the status of API packages.

Level	Explanation
1	Complete (Fully Implemented, Reviewed, Tested)
2	Mostly Complete (Mostly Implemented, Not reviewed, Not Tested)
3	Partially Complete (Most APIs covered, holes exist, not reviewed, not tested)
4	Some Implementation (Minimal functionality provided)
5	Implementation Template Classes Created
6	No Implementation

Table 1 Status Guidelines

3.2 DASE API Implementation Status

The NIST implementation adheres to the DASE API specification Version A-100 Release 0 distributed on or about June 29th, 2000. It is important to note that this is not the latest release. The current release is Version A-100 Release 4 distributed on or about October 23rd, 2000. NIST chose to freeze the implementation at this release to avoid excessive updating and changes to the implementation that would hinder progress. NIST will update the implementation when a stable (approved) version of the specification is available. Table 2 gives an overview of the API package specification to implementation mappings. The table also gives an indication of the completeness for each package.

Specification Package	Implementation Packages	Level
javax.tv.carousel	javax.tv.carousel, gov.nist.hwabstract	3
javax.tv.graphics	javax.tv.graphics	1
javax.tv.locator	gov.nist.locator	3
javax.tv.media		4
javax.tv.media.protocol		4
javax.tv.net	javax.tv.net	5
javax.tv.service	gov.nist.service	3
javax.tv.service.guide	gov.nist.service.guide	3
javax.tv.service.navigation	gov.nist.service.navigation	3
javax.tv.service.selection	gov.nist.service.selection	2
javax.tv.service.transport	gov.nist.service.transport	4
javax.tv.util		5
javax.tv.xlet	gov.nist.hwabstract	2
org.atsc.application	org.atsc.application, gov.nist.application, go.nist.hwabstract	2
org.atsc.carousel	org.atsc.carousel	1
org.atsc.data	gov.nist.data	3
org.atsc.graphics	org.atsc.graphics	5
org.atsc.management	org.atsc.management, gov.nist.hwabstract	3
org.atsc.net	org.atsc.net	5
org.atsc.preferences	org.atsc.preferences, gov.nist.preferences	2
org.atsc.registry	org.atsc.registry, gov.nist.hwabstract	2

org.atsc.security	org.atsc.security	5
org.atsc.si	gov.nist.service	5
org.atsc.si.descriptor		5
org.atsc.system	org.atsc.system	1
org.atsc.user	org.atsc.user, gov.nist.user	2
org.davic.media		4
org.davic.resources		5
org.havi.ui		4
org.havi.ui.event		4

Table 2 PAE API Implementation Status

4. DEVELOPMENT ENVIRONMENT

It is important to note the development environment for the various software packages provided in the NIST DASE Development Environment (Table 3). The STB simulation environment is not restricted in any significant way to the use of software. The DASE Applications are restricted to the DASE API specification and Java Swing. It is anticipated that the necessary Java Swing packages for an application will be downloaded as part of the application. The API implementation is restricted to the environment imposed by the DASE PAE.

General	**Specific**
Hardware	Intel Based PC Platform
Operating System	Solaris, Linux
Simulation Platform	JDK 1.2, GNU C version 2.8.1
API Implementation Platform	*Personal Java 1.2, JMF 1.1
Additional DASE Application Libraries	Java Swing

Table 3 Development Environment

*Currently the API implementation only uses classes defined in the DASE-J spec (pJava plus some additional classes). However, the entire NIST platform (API implementation plus simulation) is built and run using the JDK version 1.2 which is not pJava only. Specifically, the simulation platform makes use of classes that are not found in DASE-J. NIST has produced its own internal version of the DASE-J libraries for testing the API implementation.

The simulation environment uses a C program to initialize the JVM. The Java Native Interface (JNI) calls are based on the Java version 1.2 libraries.

In addition to Solaris and Linux, there have been numerous ports by others of the development environment to the Window NT operating system.

5. SYSTEM DESIGN

5.1 Introduction

The modularization of the DASE architecture allows for stand-alone components to be built independently. The NIST environment takes advantage of this and is implementing the complete infrastructure to develop and test a DASE PAE. The scope of the NIST environment includes implementation of the DASE API and associated managers. NIST is also developing a STB simulation platform that provides an underlying support environment for the API implementation. Together, the API implementation and simulation provide a software development environment where the DASE Java APIs can be exercised and DASE procedural applications can be tested. NIST uses the Sun Microsystems' JVM as an implementation of the AEE. Currently the NIST environment does not include an implementation of the Declarative Application Environment (DAE), although future work may include this segment.

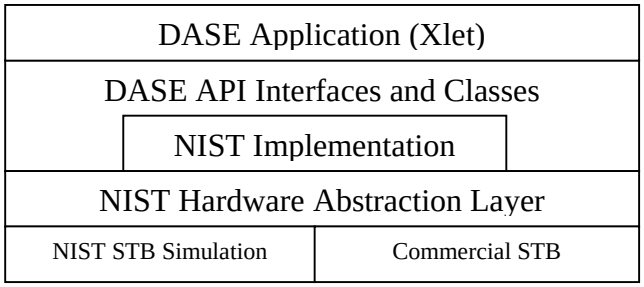


Figure 1 NIST/DASE Prototype Implementation Software Stack

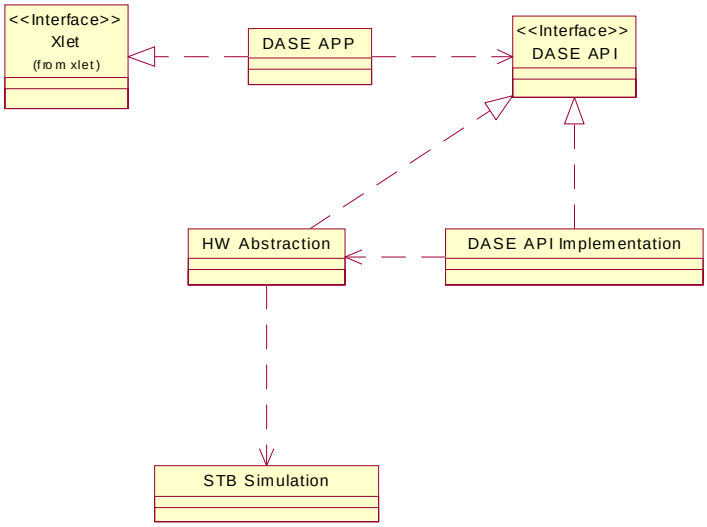


Figure 2 DASE Application Interaction

An overview of the NIST prototype implementation stack is shown in Figure 1. At the top of the stack are Xlets that access the DASE API for receiver services. Implementation classes and the Hardware Abstraction Layer fulfill the requirements of the DASE API. The HAL provides an abstraction layer between the implementation and the underlying set-top box environment.

5.2 NIST DASE Development Environment Components

5.2.1 Introduction

The class libraries of the NIST DASE RI include the following:

- **The STB Simulation Classes** - These are Java classes that implement part of the STB's functionality, such as ATSC data management, user information storage, the data carousel, etc.

These classes provide services to their clients (The Hardware Abstraction) without dependencies on the clients and are located in packages `stb.managers, stb.datatypes` and `stb.utils`.

- **The Hardware Abstraction Classes** - These classes are clients of the STB classes and provide an interface to the API implementation classes. Functions include merging of the ATSC tables, user management, and application management. Several of the DASE interfaces are implemented here. These classes are located in package `gov.nist.hwabstract`.

- **NIST DASE API Implementation Classes** - Implement the interfaces defined by the DASE API specification. These classes are contained in packages under `gov.nist`.

- **DASE API Classes and Interfaces** - These are defined by the DASE specification and include `javax.tv` and `org.atsc` packages. The interfaces defined in these packages are implemented in the NIST API packages.

- **Non-Java Modules** - These are modules written in a language other than Java, but will be accessed through Java wrapper classes. One example is the NIST ATSC transport stream parser.

In addition, the NIST DASE RI uses several other class libraries that are not part of the NIST distribution, but are expected to be part of the user's environment along with the JVM:

- **P-Java API Classes** - These are Java classes specified by DASE as part of the DASE-J (Java) run-time environment; i.e. `java.io, java.awt`, etc. The DASE API can make use of the standard Java libraries to implement API functionality. The P-Java API, as defined by Sun, is to be used.

- **Java Runtime Classes** - These are Java classes delivered as part of the Java development kit. The NIST RI uses the Sun JDK based on the Java 2 platform. Care must be taken that these class libraries are not part of the CLASSPATH visible to DASE applications.

It is anticipated that the API implementation classes will be ported to emulated and real-time environments. Therefore, the interface between the API implementation and the HW abstraction is restricted to Java method calls. However, the interface between the HW abstraction and STB functionality provided in the emulated environment may be written in Java, wrapped by a Java class, or accessed via the Java Native Interface (JNI).

One of the requirements on the implementation is that the Java classes that implement the DASE API will only call other Java class methods and not interface directly to non-Java modules. A major design goal is to have the API implementation independent of the underlying data stream encoding, while minimizing the number of classes that need to change for porting. To achieve this goal, the NIST RI is implemented as three layers with differing roles.

5.2.2 Overview of the STB Simulation

The STB layer implements functions such as ATSC stream parsing, ATSC table management, and other data management including User and Common preferences; no semantic meaning on the data is provided by the STB layer, only consistent access to the data.

A central task of the Java STB simulation classes is to provide the implementation with ATSC data structures and associated data management functions. A key aspect of the API prototype implementation design is the intermediate software HAL. It provides an interface to the STB environment that hides the details of the underlying architecture from the implementation. The HAL assumes no intelligence at the STB interface and accesses the raw MPEG/ATSC table information. At the API interface, the HAL provides a consistent view of the MPEG/ATSC table information in a manner that reflects the API definition. It is envisioned that this multi-layered design will ease the task of porting the implementation to other receiver platforms. Thus a porting effort would be focused on the HAL, which provides a central location where system level dependencies are isolated at the cost of an extra software layer. This may hinder performance in response time sensitive components of the system. However, critical performance locations can be identified and re-coded to achieve performance requirements. This layered approach is a design trade-off in the NIST implementation, which emphasizes clarity and portability over performance and efficiency

The NIST STB simulation is a collection of Java classes that encapsulate the functions of a generic ATSC STB. These classes are provided to the API implementation as services. A special class within the STB simulation manages and controls access to this Java-based simulation. The STB simulation is composed of two modules, the STB simulation control and the Java simulation classes. The simulation control boots the STB simulation and performs system initialization. Tasks include managing the JVM, running applications, and creating simulation and HAL managers.

The Java simulation classes are largely composed of manager and datatype objects. The manager objects maintain the ATSC/MPEG table data as well as STB functionality. The datatype objects are constructs that represent ATSC/MPEG table data, as well as data carousel objects. For example, an `ATSCVirtualChannel` object contains the information for a virtual channel. The datatype objects act as simple data repositories that map ATSC data into Java types. The Java simulation also includes classes to read the raw ATSC ancillary data extracted from a bitstream. The NIST STB simulation does not process the MPEG transport stream, however, but relies on an external ATSC/MPEG parser program to extract the ATSC data tables from that stream and provide them as input to the simulation. The format of this simulation input has been defined by NIST and contains only updated versions of the ATSC and MPEG tables, with no audio or video streams present.

In summary, the design philosophy with regards to the three layers is that the STB simulation performs data management by providing a repository for ATSC and other data. The HAL provides information management by creating DASE objects from the data (DASE Services from ATSC virtual channels, for example) The API implementation satisfies the DASE specification via interfaces to the HAL.

5.2.3 Hardware Abstraction Layer Overview

The Hardware Abstraction Layer merges ATSC tables (providing a single view across all of the tables), and provides the API implementation access to the data in a convenient format. All of the changes necessary to port the API should be accomplished within the HW abstraction layer. Therefore, the API implementation classes will only interface to Java classes and will retrieve all data as Java types.

The HAL Data Manager (class `DataManager`) object manages the global view of the ATSC data and other data (e.g. Users) for the API implementation. Many of the other classes within the hardware abstraction layer provide a mapping of the data from the STB into a format required by the API implementation.

Some classes in the HAL implement several of the DASE interfaces. For example, class `XletManager` in the HAL implements the `ApplicationRegistry` interface.

The Hardware Abstraction Layer is not completely void of dependencies on the API classes. In several cases, the HAL classes will implement interfaces defined by the API, such as the user registry. These HAL classes will retrieve the underlying data from the STB simulation. When a port is done, the HAL classes will need to change the low-level access to the data, but their public interfaces will not change. When data must be accessed with a single view (such as user management), the HAL performs this data aggregation.

5.2.4 DASE API Overview

The API implementation classes comprise the third layer. Semantic rules defined in the DASE API specification (such as user preference filtering) will be done in the API implementation classes.

The DASE API functionality is provided by a set of Java classes that implements the API methods. The naming conventions we have chosen to use are as follows: API classes and interfaces are named exactly as they appear in the DASE specification. Implementation classes that support an interface use the interface name with `Class` appended. For example, the API class `Service` is implemented by class `ServiceClass`.

The core work of the implementation classes is to map the data maintained by the hardware abstraction layer into the views provided by the DASE API. For example, the implementation class `ServiceCollectionClass` will use data from the ATSC virtual channel table, as well as other ATSC tables, to return collections of `Service` objects. Other functionality of the API implementation is to provide for control over streaming data by the DASE application.

In summary, the design philosophy with regards to the three layers is this: The STB simulation provides *data* management by providing a repository for ATSC and other data. The HAL provides *information* management by creating DASE objects from the data (DASE Services from ATSC Virtual channels, for example). The API implementation fulfills the DASE specification and interfaces to the HAL. Most of the behavior defined in the DASE specification is implemented here, although in some cases that behavior will be implemented by the HAL. For example, the Xlet lifecycle model is implemented in the HAL.

6. THE STB SIMULATION

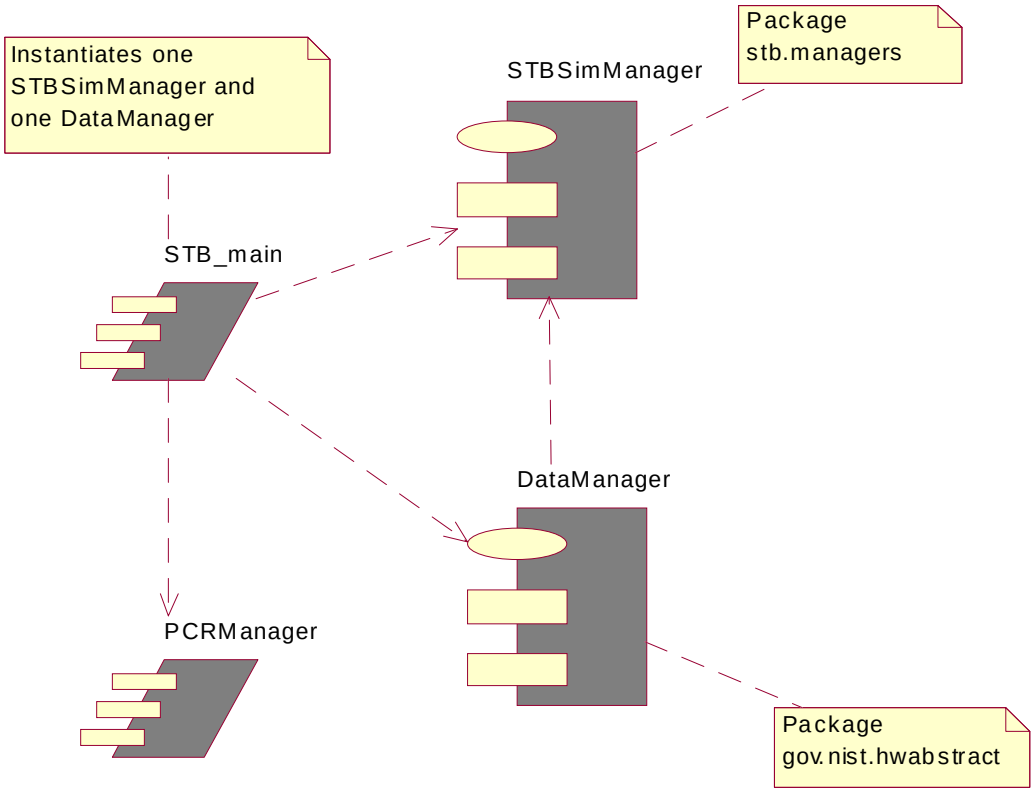


Figure 3 Simulation Components

6.1 Introduction

The STB simulation provides the platform on which the DASE API executes. Figure 2 shows the Unified Modeling Language (UML) diagram for the DASE application to API interaction. The diagram shows how the API implementation (supporting the interface defined by DASE) relies on the hardware abstraction and STB simulation to provide services.

Figure 3 shows the main components of the STB simulation and the runtime relationships between the components. The controlling process for the simulation is a C program called `STB_main`. This program starts up the Java virtual machine (JVM) and creates the STB manager (class `STBSimManager`) and the HAL `DataManager` objects within the JVM. Also, `STB_main` creates the `PCRManager` program as a child process.

6.2 The Native Code Components

6.2.1 The Simulation Control Program

The central point for starting and controlling the STB simulation is the program `STB_main` that is written in the 'C' language. This program performs several functions:

- Loads the Java Virtual Machine (JVM)
- Controls the logging process for itself as well as the JVM and Java code
- Starts the Program Clock Reference (PCR) generator program
- Allows for the launching of Java programs from the command line

`STB_main` loads the JVM upon startup. Two objects are instantiated within the JVM with the logging option set based on the command line entry. These objects are `STBSimManager` and `DataManager` as can be seen in Figure 3. If the attempt to create either of these objects fails, the simulation terminates.

The PCR generator program, `PCRManager`, is a stand-alone program started as child process by `STB_main`. The command line parameters are passed to `PCRManager` from `STB_main`, notably the socket number used by the program to communicate with the HAL. The next section describes `PCRManager` in more detail.

The NIST DASE Development Environment User's Guide [USER-GUIDE] provides detailed instructions on running the simulation.

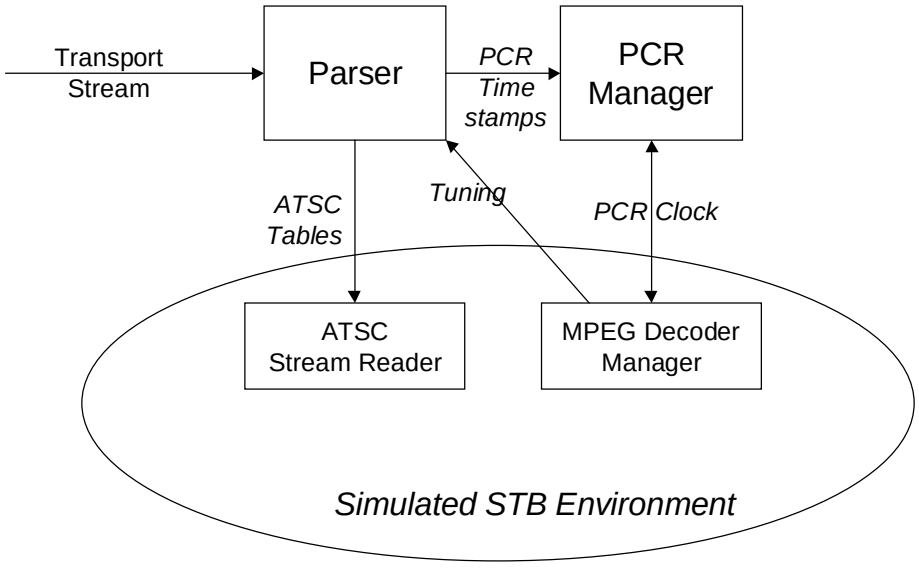


Figure 4 Decoder Hardware Simulation

6.2.2 Program Clock Reference Manager

This component simulates the real-time hardware PCR that should be found on any hardware MPEG decoder. Figure 4 shows the interactions between the PCR Manager and the rest of the simulation environment. The PCR Manager acts as the interface between the stream parser that periodically receives PCR timestamps and software components higher in the rendering chain that want to read such a clock on a random-access basis.

The specification for ATSC requires that PCR timestamps be sent at least every 20 ms, so the PCR server does not implement a complex phase-locked loop (PLL), but rather a simple one-time linear adjustment algorithm.

6.2.2.1 Stand-alone mode

Syntax of the program (use -h for details):

```
PCRManager [-p port] [-w file] [-f fifoName] [-i PID] [-v] [-v  -i PID .....]
```

`PCRManager` tracks the local clock on the PID number defined by DSTP_VIDEO_PID (currently 0x8951=35153). If no FIFO is specified on the command line, this PID can be used for test purposes, without a parser attached. Upon receipt of a SIGUSR2 signal, a discontinuity condition is simulated on the test stream. A SIGHUP signal will reset the clock to 0 with a discontinuity condition.

Additional PIDs may be simulated with the `-i` option; they are not affected by signals.

6.2.2.2 Interface to the Stream Parser

The parser sends each PCR Timestamp it receives to the `PCRManager`. Theoretically, there may be as many as 64K different PCR clocks, one for each possible PID. Although it is most likely that a hardware device will track only one PCR clock (which one?), `PCRManager` will track any number of PCR streams. For each PID encountered in the stream, the manager maintains a clock trained to the timestamps received from the parser; it also keeps track of discontinuity conditions indicated in the header (a counter is incremented for each discontinuity). The structure of a PCR packet from the parser is shown below:

```
struct PCRTimeStampStruct {
  unsigned char header;
  unsigned char flags;
  unsigned short PID;
  unsigned int PCRHigh;
  unsigned int PCRLow;
  unsigned int padding;
};
```

6.2.2.3 Interface to Clients

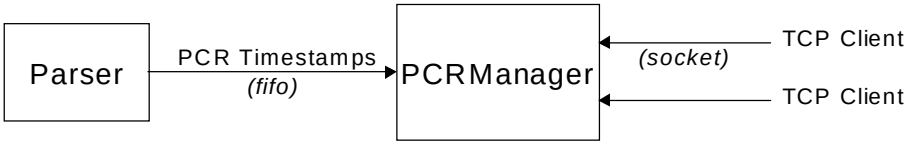


Figure 5 PCR Manager to Client Communication

The interface with rendering components is done concurrently via TCP/IP sockets. Figure 5 shows the PCR Manager interaction with a TCP/IP client. This model is intended to simulate the behavior of a device driver (many being able to access it at the same time). A client will connect to the port specified in the command line. The communication between both ends is done via standard packets shown below:

```
struct PCRSlaveRequestStruct {
  unsigned char PIDHigh;                /* 1 */
```

```
  unsigned char PIDLow;                 /* 1 */
  unsigned char pcr[8];                 /* 8 */
  unsigned char continuityCounter;      /* 1 */
  char invalid;                         /* 1 */
};
```

A client prepares a packet by filling the PCR field. Upon receipt, the server fills in the other fields, or sets the invalid flag, and sends the message back.

6.2.2.4 Test Feeder

`PCRTestFeeder` is a test program to generate parser packets.

6.2.3 The Bitstream Parser

The parser process demultiplexes the transport stream from a file or a pipe. It routes the ATSC tables to a reader object in the Simulation (described elsewhere). PCR Timestamps are sent to a `PCRManager` process (see below). A back channel to the parser is available for simple tuning operations.

6.3 Java Components

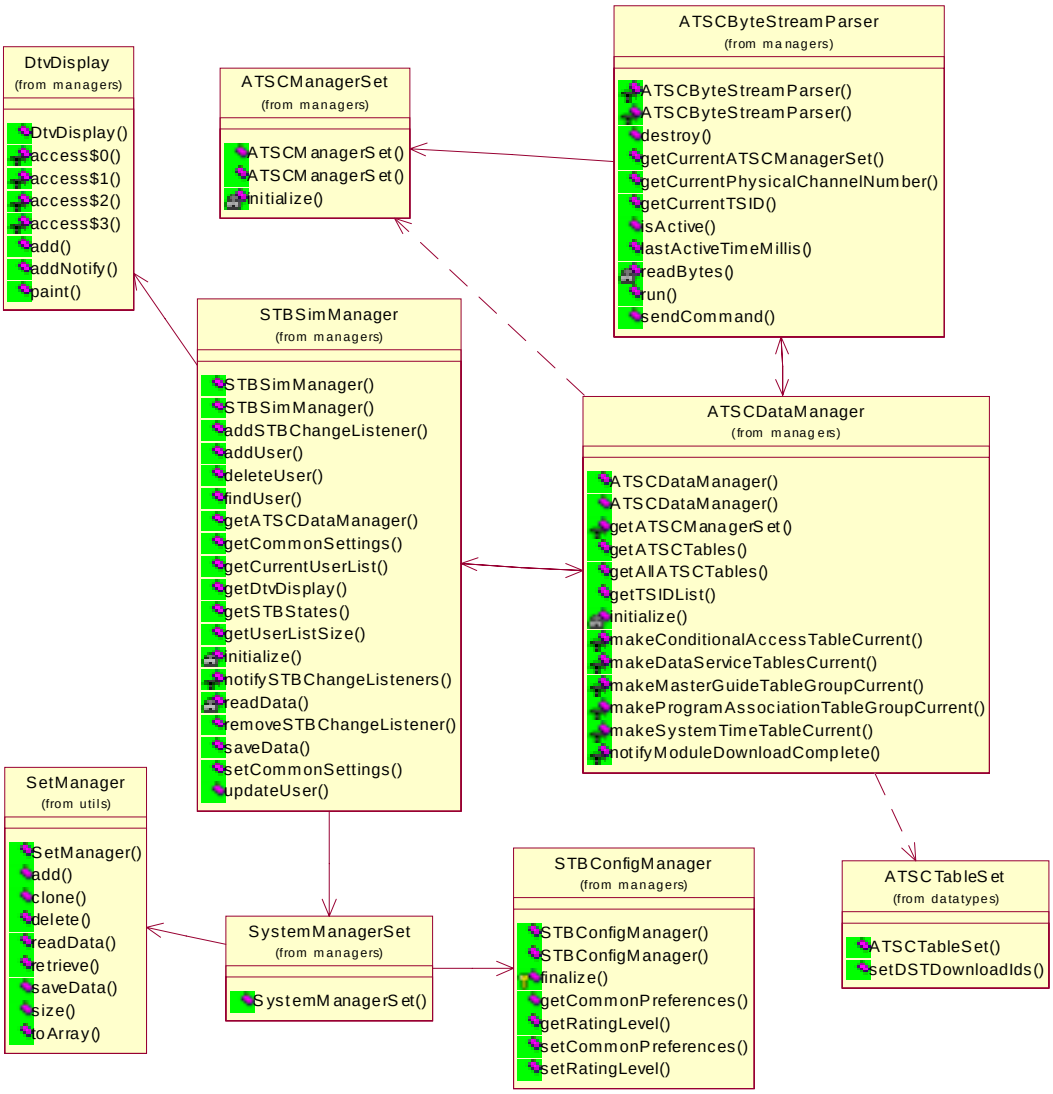


Figure 6 Simulation Management Classes

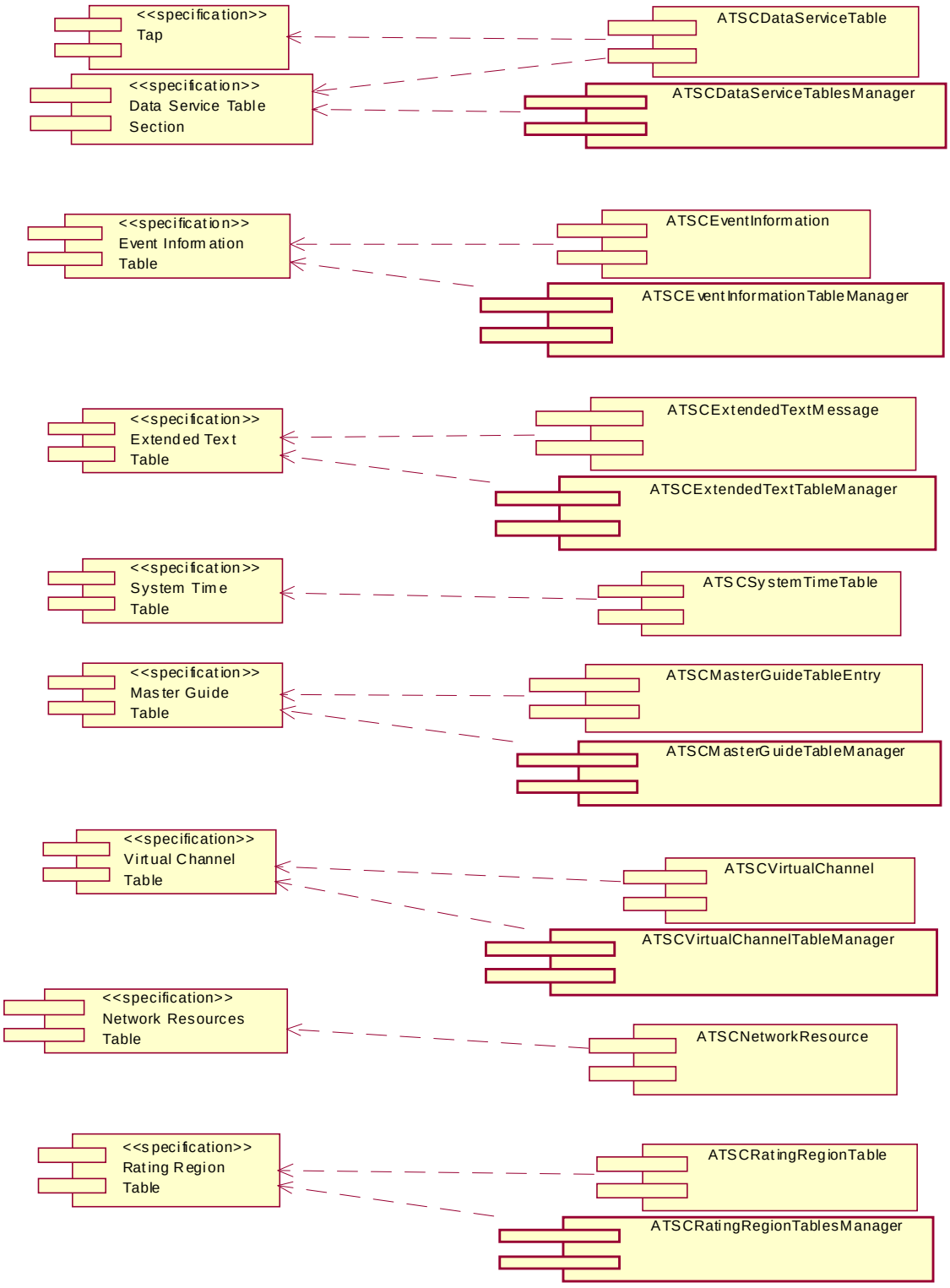


Figure 7 Data Management Components

Figure 6 gives the class diagram for the simulation classes used to maintain the STB state. For example, configuration management, Figure 7 shows the mapping of the ATSC data tables to the simulation components. The components in right side of the diagram represent the Java classes used to store and manage the ATSC table data. The data and management classes are discussed in the next sections.

6.3.1 The Manager Classes

The manager classes are responsible for maintaining the ATSC data tables as well as users, application data, and other data sets associated with the STB simulation. These classes are in package `stb.managers`.

The simulation manager `STBSimManager` object is responsible for creating the ATSC table manager objects, the ATSC data stream readers `ATSCByteStreamParser` (one for each virtual tuner), and several other managers such as the STB configuration manager. The `STBSimManager` implements several management functions directly, such as user management.

Each table manager handles one type of ATSC table, and only one instance of the table. In order to manage more than one table instance (such as when there are multiple streams) multiple instances of the table manager must be created. Examples of table managers shown in Figure 7 are the `VirtualChannelTableManager` and `MasterGuideTableManager` classes.

The highest visibility of the ATSC table manager classes is **package**. This restriction means that only classes contained within the `stb.managers` package can call the retrieval methods. The ATSC tables are private data within the manager classes. Therefore, the retrieval methods of each manager return a reference to an array containing the data elements of that table. There is no danger of having the array updated (due to the arrival of a revised table in the input stream) during the `getXxx()` method because any new updates to the table are maintained in a separate array, and access to the current arrays is synchronized.

Table managers are grouped into sets where one set manages one transport stream (as identified by its TSID). The `ATSCManagerSet` class maintains this grouping.

Class `ATSCDataManager` acts as a central control point for access to all of the ATSC and MPEG tables. A reference to an object of this class is returned by method `STBSimManager.getATSCDataManager()`. The public methods of class `ATSCDataManager` provides access to the ATSC tables and MPEG PMT/PAT tables via the ATSC table managers. The tables are returned as a set with references to the tables contained in an `ATSCTableSet` object.

6.3.2 Input Stream Parsers

The parser classes in package `stb.managers` are responsible for extracting the information from the MPEG and ATSC tables, as well as Data Carousels, contained within the NIST-defined input stream. These classes cannot be used to parse an MPEG transport stream as the simulation relies on an external process to perform that function.

Class `ATSCByteStreamParser` is the main parser for the ATSC data stream format defined by NIST (see [USER-GUIDE] for a description of this stream). This class extends the `Thread` class in order to run independent of the main STB simulation. The simulation manager `STBSimManager` creates an object of this class as an active thread.

The `run()` method of this class waits to read data over a FIFO and determines what type of data it is (ATSC PSIP table, MPEG table, Data Carousel, etc.). `ATSCByteStreamParser` calls method `parseSection()` of an object of class `PrivateSectionParser` to perform the actual data stream parsing.

6.3.3 The Datatype Classes

```
package stb.datatypes;
public class ATSCVirtualChannel extends ATSCDatatype implements Cloneable {
  /** Indicates if this is a terrestrial or a cable virtual channel. */
  public int               deliverySystemType;
  /** Constant for <code>deliverySystemType</code>. */
  public final static int  DELIVERY_TERRESTRIAL = 1;
  /** Constant for <code>deliverySystemType</code>. */
  public final static int  DELIVERY_CABLE       = 2;
  /** Constant for <code>deliverySystemType</code>. */
  public final static int  DELIVERY_SATELLITE   = 3;
  public String    short_name;
  public short     major_channel_number;
  public short     minor_channel_number;
  public short     modulation_mode;
  public long      carrier_frequency;
  public int       channel_TSID;
  public int       program_number;
  public byte      ETM_location;
  public boolean   access_controlled;
  public boolean   hidden;
  public boolean   hide_guide;
  public boolean   path_select;
  public boolean   out_of_band;
  public byte      service_type;
  public int       source_id;
  /**
   * <code>ATSCArrayList</code> of <code>ATSCDescriptor</code>s.
   */
  public ATSCArrayList    descriptors = new ATSCArrayList(5);
  /**
   * Creates and returns a deep copy of this object.
   * @return A deep copy of this instance.
   */
  public Object clone() {
      :
  } /* public Object clone() */
} /* public class ATSCVirtualChannel */
```

Figure 8 Class ATSCVirtualChannel

The datatype classes correspond to the entries in the ATSC data tables and other information managed by the STB simulation (Users, Preferences, etc.) These classes are in package `stb.datatypes`. For example, an `ATSCVirtualChannel` object contains the information for one virtual channel. The datatype classes act as simple data repositories for the ATSC data mapped into Java types.

All of the ATSC data classes are derived from a base class called `ATSCDatatype` that implements the *Cloneable* and *Serializable* interfaces to enable deep copying and object serialization. The non-ATSC

data classes (`User` for example) derive from base class `SimDatatype` that also implements *Cloneable* and *Serializable*.

The datatype classes define all of the attributes of the class as **public**. Therefore, there are no access methods; the data items are manipulated directly. This approach simplifies the client programming in that there is no need to call several methods to retrieve simple data items. The attributes of the class have the same names as the tables defined in [ATSC:A65]. The simulation assumes that the client classes will access the data in a read-only fashion. Therefore, if a client class changes the data fields, that change will be reflected to other clients because references to the data are returned from the ATSC table managers, not clones. The philosophy of the simulation is to treat the data as a shared memory segment would be treated, with little locking. However, locking of the tables does occur when the `makeCurrent()` method of a table manager is called because this method is declared `synchronized`. This locking mechanism allows the current version of the table to be replaced with a new version and prevents clients from retrieving a mix of current and new data.

Figure 8 shows the ATSC Virtual Channel class as an example. The descriptors associated with the table are included in the class as an `ATSCArrayList`, a datatype class created for consistent access to ATSC descriptors. Also, the `clone()` method is overridden in classes where it is necessary to provide deep copying.

Other classes contained in package `stb.datatypes` are classes used to represent various data items internal to the simulation. Examples of these classes include `CommonSettings` to store the STB settings and preferences, and various exceptions thrown by the STB simulation.

6.3.4 Other Classes

These classes are various stand-alone utilities like Huffman coder/decoder, output functions in compressed bitstream form and logging classes used throughout the simulation. Class `ATSCInputStream` contains utility methods used to parse specific sections of the input stream. The classes are in package `stb.utils`. Another class in this package, `SetManager`, implements a simple associative set, useful for small databases. The simulation manager uses this class to create databases of users, STB settings, and other set-top box information.

6.4 PSIP Table Management in the Simulation

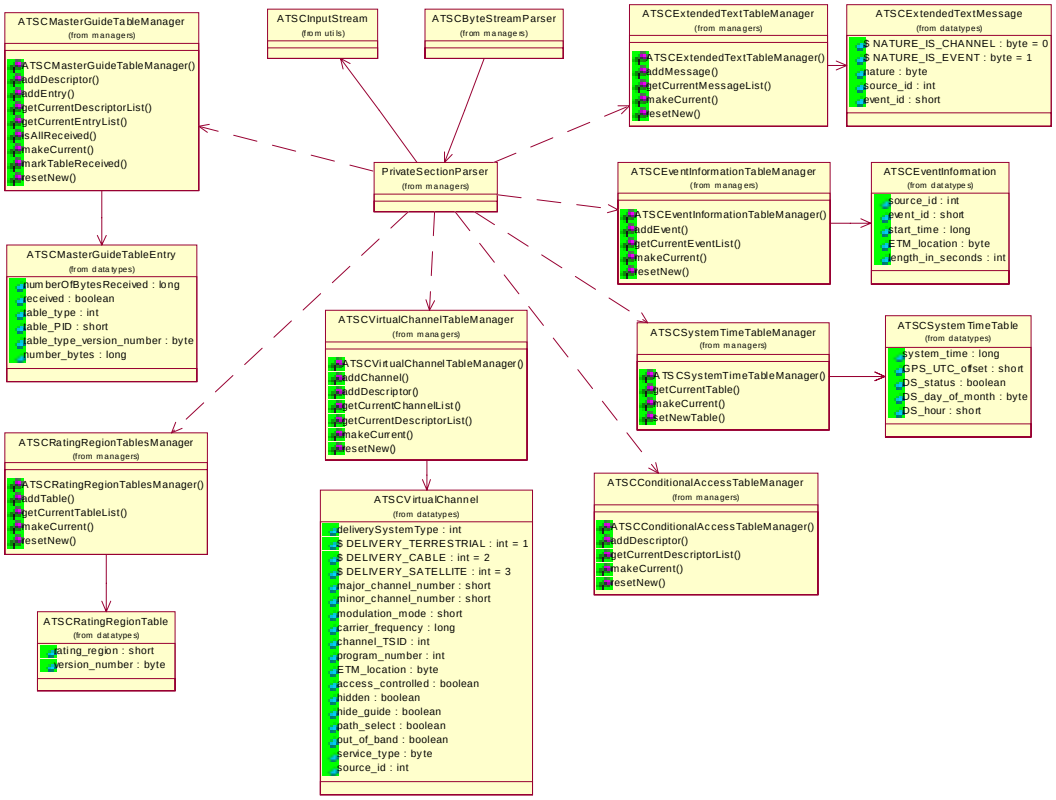


Figure 9 PSIP Management Classes

6.4.1 PSIP Table Classes

This section describes the management of the Program and System Information Protocol (PSIP) tables within the simulation. The definition of the PSIP tables is contained in [ATSC:A65]. The simulation classes directly map the PSIP tables in terms of data layout and management as described in Section 6.3. This section provides more detail on the classes used to maintain and manage the PSIP table data. Figure 9 shows the class diagram for PSIP management.

What follows is a brief description of the classes used to manage the PSIP tables within the simulation. Because table management is a core issue in any STB, these classes may be reused and recombined to support the needs of PSIP data.

6.4.1.1 Manager Classses

These classes are located in package `stb.managers`.

6.4.1.1.1 ATSCDataManager

This class is the main access point for all of the table manager classes. It provides synchronized access to the tables so that a complete table will be accessed, and also that the complete set of tables associated with the Master Guide Table are kept in alignment.

6.4.1.1.2 ATSCEventInformationTableManager

It manages a collection of events as defined in [ATSC:A65]. This class is analogous to the Event Information Table.

6.4.1.1.3 ATSCMasterGuideTableManager

Manages a collection of entries from the Master Guide Table (MGT). This collection includes all descriptors sent down with the MGT.

6.4.1.1.4 ATSCRatingRegionTablesManager

Manages a collection of Rating Region Tables (RRT). Each table contains the rating information for a single rating region.

6.4.1.1.5 ATSCVirtualChannelTableManager

This class manages the collection of virtual channels. A current collection and new collection are maintained, and when a new table is parsed, the new collection replaces the current collection.

6.4.1.1.6 ATSCExtendedTextTableManager

This class manages a collection of extended messages.

6.4.1.2 Datatype Classes

These classes are located in package `stb.datatypes`.

6.4.1.2.1 ATSCDatatype

This class is the parent class of all the datatype classes.

6.4.1.2.2 ATSCEventInformation

This class contains the information for one event of an Event Information Table, as defined in [ATSC:A65]. It is a simple repository for the event information by mapping the event information into Java types.

6.4.1.2.3 ATSCExtendedTextMessage

This class contains the information for a single Extended Text Message, as defined in [ATSC:A65]. It is a simple data repository for the Extended Text Message data as Java types.

6.4.1.2.4 ATSCMasterGuideTableEntry

This class contains the information for one entry of the Master Guide Table, as defined in [ATSC:A65].

6.4.1.2.5 ATSCRatingRegionTable

This class contains the information for a Rating Region Table, as defined in [ATSC:A65].

6.4.1.2.6 ATSCVirtualChannel

This class contains the information for a single virtual channel, as defined in [ATSC:A65]. Virtual channels from both the Cable Virtual Channel Table (CVCT) and Terrestrial Virtual Channel Table (TVCT) are represented by this class.

6.4.2 Virtual Channel Table Example

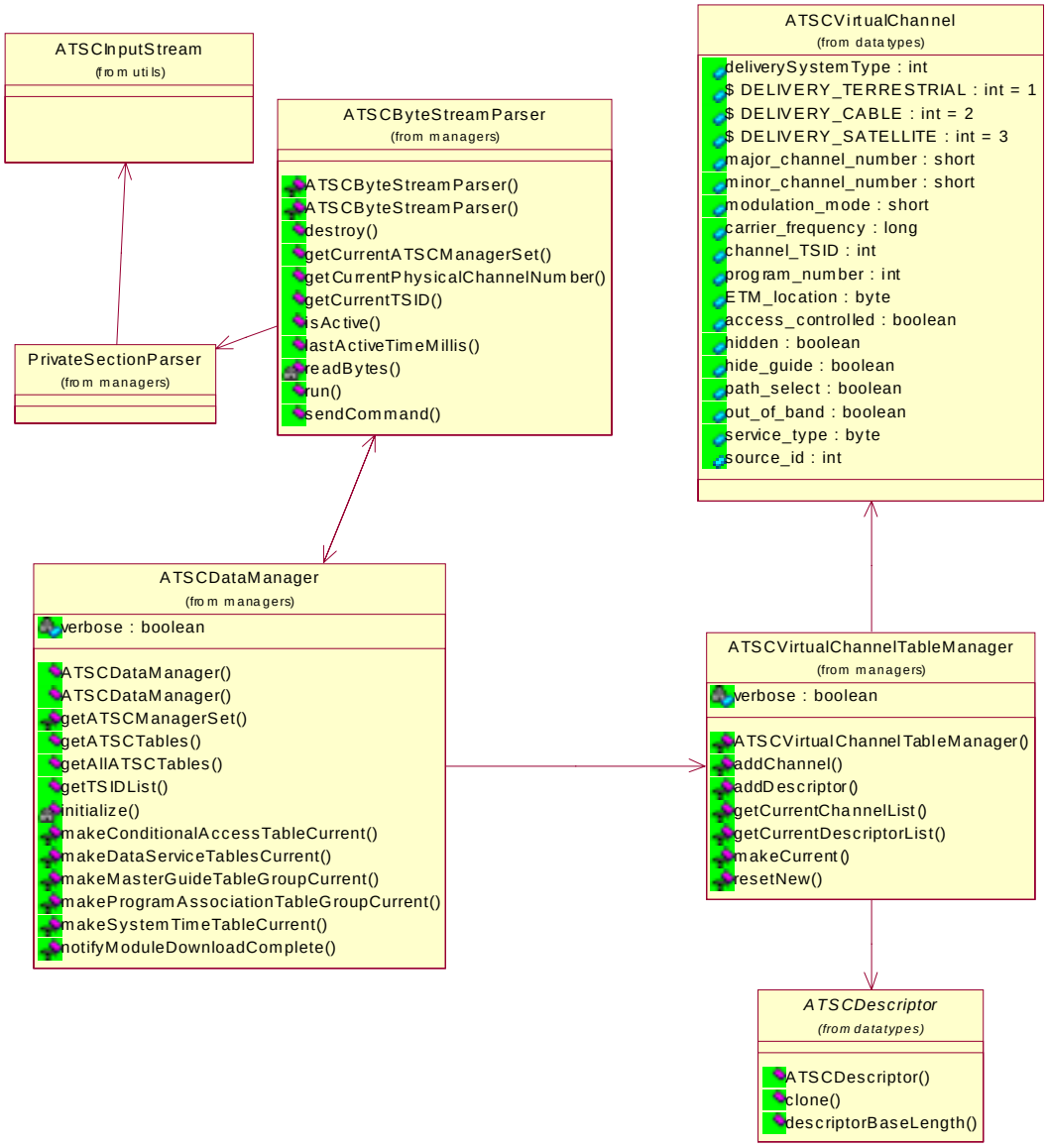


Figure 10 Virtual Channel Table Management

Figure 11 Virtual Channel Parsing

As stated above, the main function of the STB Simulation is to provide simplified access to the ATSC and MPEG tables. The simulation maps the data from the bitstream into Java classes and provides synchronized access to this data. This section discusses how the ATSC and MPEG tables are parsed and how the data is then managed. The parsing and managing of the ATSC Virtual Channel table is used as an example to trace the flow of data through the simulation.

Figure 10 shows the relationship between the classes used for ATSC Virtual Channel Table parsing and management. Figure 11 is the interaction diagram showing the method invocations needed to perform virtual channel parsing.

The `ATSCByteStreamParser` object reads the bitstream from the FIFO and calls method `parseSection()` of `PrivateSectionParser` to parse virtual channels from the bitstream by using an `ATSCInputStream` object. `PrivateSectionParser` adds the virtual channel objects to the `ATSCVirtualChannelTableManager` by calling the `addChannel()` method. A similar procedure occurs for adding the descriptors for the virtual channels. After all of the virtual channels and descriptors are read, method `ATSCVirtualChannelTableManager.makeCurrent()` is called (not shown in Figure 11) to move the new virtual channel information to the current table. Subsequent calls to retrieve virtual channels will result in the new virtual channel information being retrieved.

When the complete table (or set of tables in the case of the Master Guide Table and all associated tables) is received, the table set manager `ATSCDataManager` is notified by `ATSCByteStreamParser` and the new table (or set of tables) is made current.

6.5 Data Broadcast Management

The STB manager is responsible for managing the resources for the data broadcast application as retrieved from the data carousel, including the Java byte codes for the Xlet classes. Class `PrivateSectionParser` retrieves the downloaded modules from the stream and adds the modules to the `ATSCUserToNetworkDownloadManager` object. See [ATSC:A90] for a description of the data broadcast standard. Downloaded Xlets and their data are transmitted using the referenced standard.

Classes in the HW abstraction layer manage the applications in accordance to the data broadcast specification as well as the DASE view, where 'application' means Xlet. In keeping with the overall design philosophy where the STB manages raw data and the HW abstraction provides semantic meaning, responsibility for Xlet management lies in the HW abstraction. Support for the DASE application lifecycle model is also provided by the HAL.

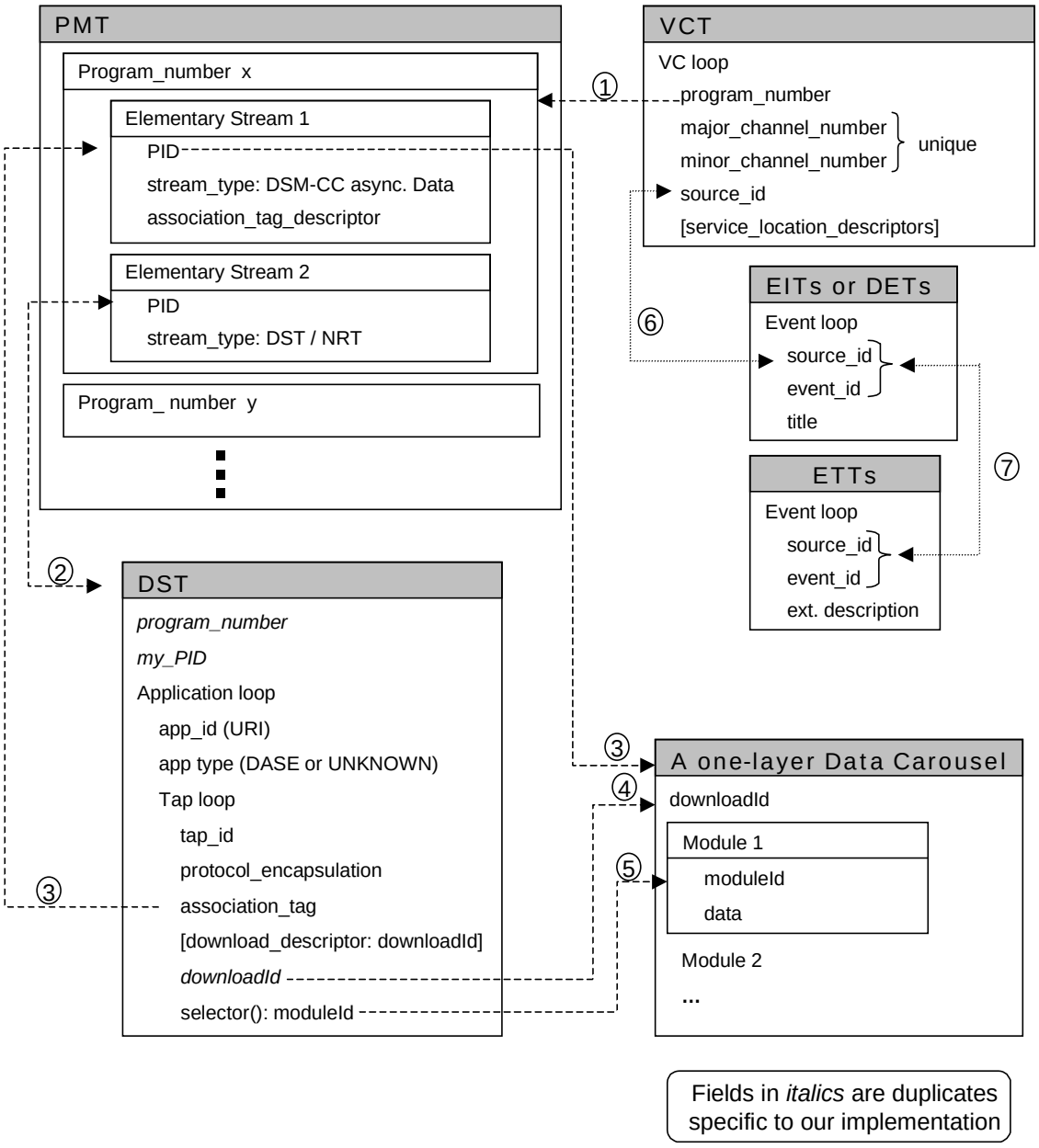


Figure 12 Data Service Discovery and Extraction

6.5.1 Data Broadcast Overview

As in the rest of the simulation, DST data structures directly map the MPEG and ATSC structures. However, three fields are duplicated in the DST class and are set at parsing time:

- The `program_number` of the PMT program containing the Data Service - This allows easy lookup when matching `association_tag`s between DST and PMT.
- The `PID` that carried the DST - Right now, this is used neither by the simulation nor by the HAL, but will be used in later versions to implement application signaling.
- In DST Taps, the `downloadId` that identifies a Data Carousel.

There are different ways to relate ATSC and MPEG tables. The relationship arrows in Figure 12 illustrate the chosen method in our implementation:

- ~~Arrow ①: In the VCT, the `program_number` field associates a Virtual Channel with a list of resources in the PMT, that is a PMT Program. We could have used the more convenient Service Location Descriptors that are mandatory for terrestrial broadcast, but since they are optional in cable broadcast, going through the PMT appeared to be a safer choice.~~

The set of MPEG-2 program map tables (PMTs) of a transport stream is stored as entries of an array. Similarly the set of virtual channels of an ATSC virtual channel table (VCT) are also stored as entries of an array. The `program_number` is the data element that relates the corresponding entries of these two arrays, as shown in relationship ① of Figure 12. In this context a (ATSC) virtual channel and a (MPEG-2) program are synonymous. Each PMT contains a list of resources (elementary streams) for the service specified in that program. Each virtual channel of the VCT contains the same list of resources for terrestrial broadcasts, but may not for cable. For this reason we chose to use the PMT for this information, and also because it contains the added data element `association_tag`, when appropriate.

- ~~Arrow ②: In the PMT, there can be at most only one DST/NRT stream per Program, as a direct consequence of a restriction to one Data Service per Virtual Channel (see [ATSC:A90]). If there is such a stream, the `PID` permits to extract it from the Transport Stream and parse the DST (and optionally the NRT which is not currently implemented in the NIST simulator).~~

A DSM-CC stream is specified by its PID data element within the list of streams of a program/virtual channel. It is also labeled by an `association_tag` in the PMT that corresponds to an `association_tag` in a Tap loop (application resources) of a DST. By searching the list of PMTs, using the `program_number` and an `association_tag` from a DST, one can locate the PID of the corresponding DSM-CC stream. This is shown in relationship ③ of Figure 12. By searching the list of DSM-CC streams, using a `downloadid` from a DST, one can also locate the corresponding DSM-CC stream. This is shown in relationship ④ of Figure 12. Specific parts of a DSM-CC stream (e.g., a module within a carousel) can be further referenced via the `selector()` field, as shown in relationship ⑤ of Figure 12. One example is for an application (service) carried in a DSM-CC stream as a carousel, where class files are carried in various carousel modules, another module may contain a jar file and still other modules may contain data files.

- ~~Arrow ③: In the DST, each Tap structure (i.e. application resource description) has an `association_tag` field. Coming back to the PMT, the `association_tag` is the key to retrieve the `PID` of the elementary stream that carries the resource.~~

- ~~Arrows ④ and ⑤: From this `PID` and optionally the `download_descriptor` (DST) we can extract the resource from the Transport Stream (e.g. a Data Carousel), or only the part we need with the~~

~~additional information contained in the `selector( )` structure of the Tap (e.g. one module of a Data Carousel).~~

Each virtual channel of a VCT also contains references to announcement tables. The Event Information Tables (EITs) and the Data Event Tables (DETs) contain event (service - program or application) information similar to a TV guide. The `source_id` of a virtual channel can be used to search all the EITs and DETs to locate events applicable to that virtual channel, as shown in relationship of Figure 12. For each event, the pair `source_id` and `event_id` can be used to search the list of Event Text Tables (ETTs) to locate further announcement information about that event, as shown in relationship ⑦ of Figure 12.

DASE currently limits the number of data services per program/virtual channel to one, but this may increase in future versions. Thus currently limiting a single data service table/ network resource table (DST/NRT) stream per PMT. The appearance of an application within a DST/NRT stream signals the initiation of a data service and the removal of that application from the DST/NRT stream signals the termination of that data service. A DST/NRT stream is specified by its PID data element within the list of streams of a program/virtual channel. By searching the list of DSTs, using the `program_number` and the PID, one can locate the corresponding DST from a PMT. By searching the list of PMTs, using the `program_number` and `my_PID`, one can locate the corresponding PMT from a DST. This is shown in relationship ② of Figure 12.

Note: The `download_id` field in the Tap structure was added as a convenient shortcut when the `download_descriptor` is not present.

Additional information about a Data Service (e.g. announcement and extended description) can be found in the EIT/DET/ETT tables. The linkage happens through a common `source_id` value. Note that this `source_id` value doesn't necessarily uniquely identify one Virtual Channel, as opposed to the `major/minor_channel_number` pair.

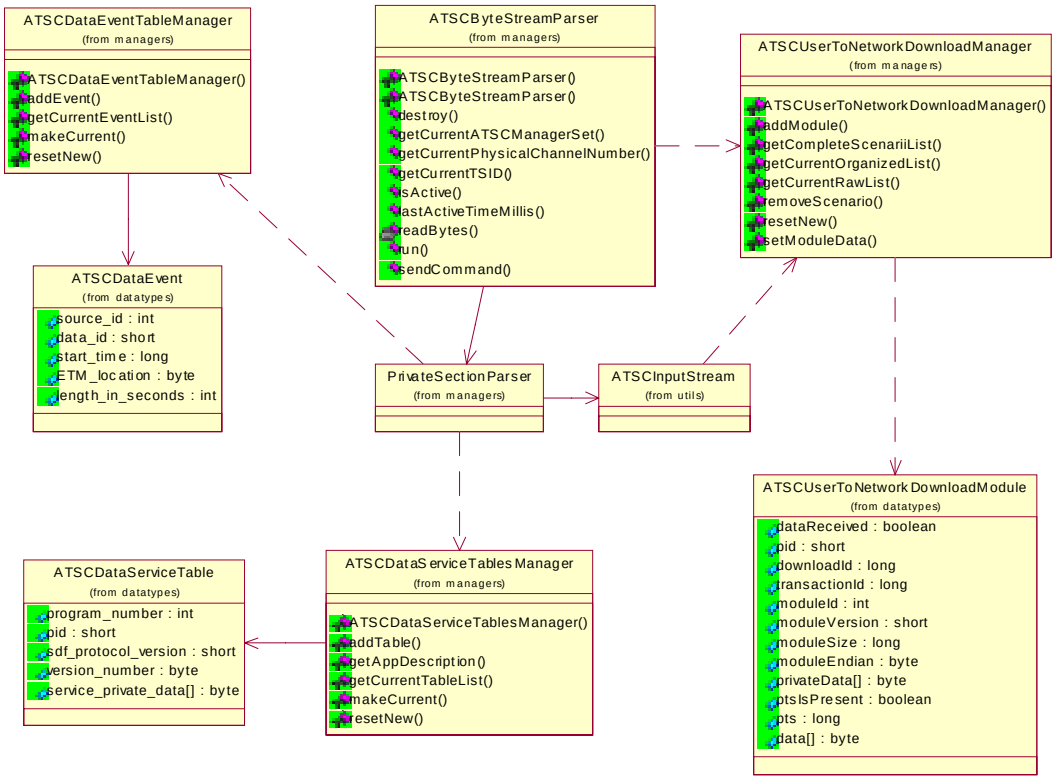


Figure 13 Class Diagram for Data Service Extraction

6.5.2 Simulation Management of Data Broadcast

Figure 13 shows the diagram for the classes involved in data service parsing within the simulation. Each downloaded module is stored within a collection inside the `ATSCUserToNetworkDownloadManager` object. Also shown are the remaining classes used to store the information from the other ATSC tables in addition to the Virtual Channel Table as discussed in Section 6.4. The definition of these classes can be found in the `stb.datatypes` package.

6.5.2.1 ATSCDataServiceTablesManager

This class, in package `stb.managers`, manages a collection of Data Service Tables. Each Data Service Table describes one data service.

6.5.2.2 ATSCDataServiceTable

This class, in package `stb.datatypes`, contains the information for one Data Service Table, as defined in the ATSC Data Broadcast standard [ATSC:A90]. One Data Service Table describes the components of one data service.

6.5.2.3 ATSCDataEventTableManager

This class in package `stb.managers` manages a collection of Data Events. It is analogous to the Data Event Table as defined by [ATSC:A90].

6.5.2.4 ATSCDataEvent

This class, in package `stb.datatypes`, contains the information for one event of a Data Event Table, as defined in the ATSC standard [ATSC:A90]. This class is a data repository only that maps the items for a data event into the Java types.

6.5.2.5 ATSCUserToNetworkDownloadManager

This class is in package `stb.managers`. It manages a module repository for the DSM-CC User-To-Network Download Scenario, as defined in the MPEG ISO/IEC 13818-1 standard and the ATSC Data Broadcast standard [ATSC:A90]. It does a different task than managing an ATSC table, therefore, does not extend `ATSCTableManager`. The functionality implemented is based on a one-layer version of the Data Carousel.

This class downloads every module of each download scenario. Each time a module (DII+DDB) has been completely downloaded, it moves from the new module repository to the current module repository.

6.5.2.6 ATSCUserToNetworkDownloadModule

This class contains the description and the data for one module of a DSM-CC User-To-Network Download Scenario, as defined in the MPEG ISO/IEC 13818-6 and [ATSC:A65] Data Broadcast specifications. This data structure is designed for merging information contained in DII and DDB messages. This is appropriate for DSM-CC Download Scenarios that exclude the possibility of a return channel (the data carousel scenario and, without return channel, non-flow-controlled download scenario). This class is part of the `stb.datatypes` package.

6.6 JMF

6.6.1 Hardware Simulation

Describe PCR Manager and tuning here.

6.6.2 MPEGDecoderManager

This standard simulation manager establishes the link between the Java Simulation environment and the various components simulating the hardware.

Class `MPEGDecoderManager` implements some of the interfaces described above. `DataManager` creates one instance at startup and makes references to it available through its `get`*`Device`*`()` method (where *`Device`* is the name of the requested interface).

Currently implemented interfaces:

- `VideoDecoder`: establishes a connection with the running `PCRManager` process and provides access to the current PCR time.
- `Demux`: tuning within a stream and low-level audio track selection, via the parser's back channel. *Not Implemented*.
- `Tuner`: *Not implemented*.
- `TimeBase`: provides access to the low-level timeline (PCR) a standard `javax.media.TimeBase` (JMF) time base. Unlike the raw PCR timeline that can contain discontinuities, the time base provided is strictly monotonic, even across discontinuities,

tuning and other interruption. However it always ticks at the same pace as the currently displayed video stream.

6.6.3 PCRManagerPullSourceStream

This class wraps the simulation/emulation components into a standard JMF object. In order to comply with the standard JMF architecture, the `read()` method returns the current PCR time serialized as 8 byte packets.

In addition to the `PullSourceStream` interface, this object also acts as a pass-through to the objects implementing the `VideoDecoder`, `Tuner`, `Demux` and `TimeBase` interfaces. If performance is an issue, it is recommended to use the `VideoDecoder` or `TimeBase` interfaces to read the current media time to avoid serialization overhead -- at the cost of portability, of course.

6.6.4 DataSource

The Java Media Framework uses a `DataSource` during the instantiation process. It is associated during initialization to the "pcr:" protocol; MRLs requesting a "pcr:" object are processed by a new `DataSource`, that parses the MRL and in turn instantiates a `PullSourceStream` accordingly. See the official JMF Documentation for details.

6.6.5 Handler

6.6.6 DAVIC Controls

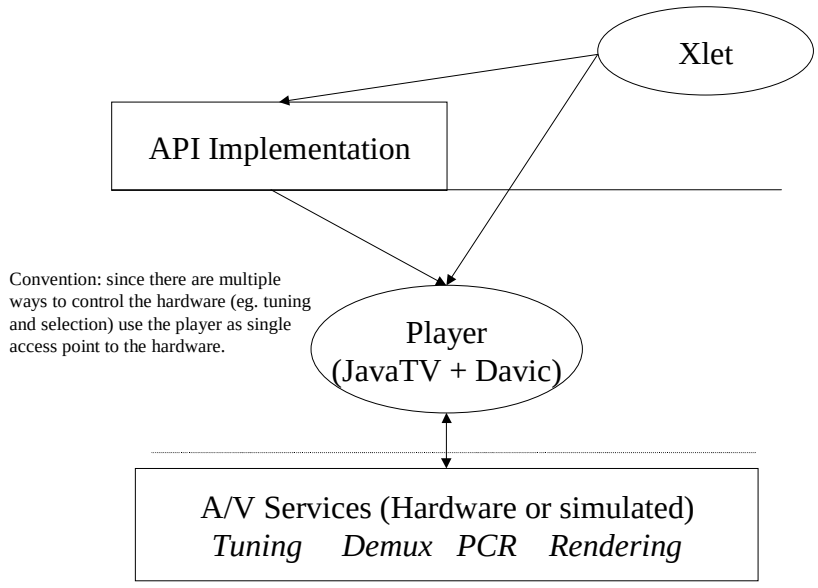


Figure 14 JMF Support and Extensions

7. REAL-TIME EMULATION

8. COMMERCIAL STB

9. HARDWARE ABSTRACTION LAYER

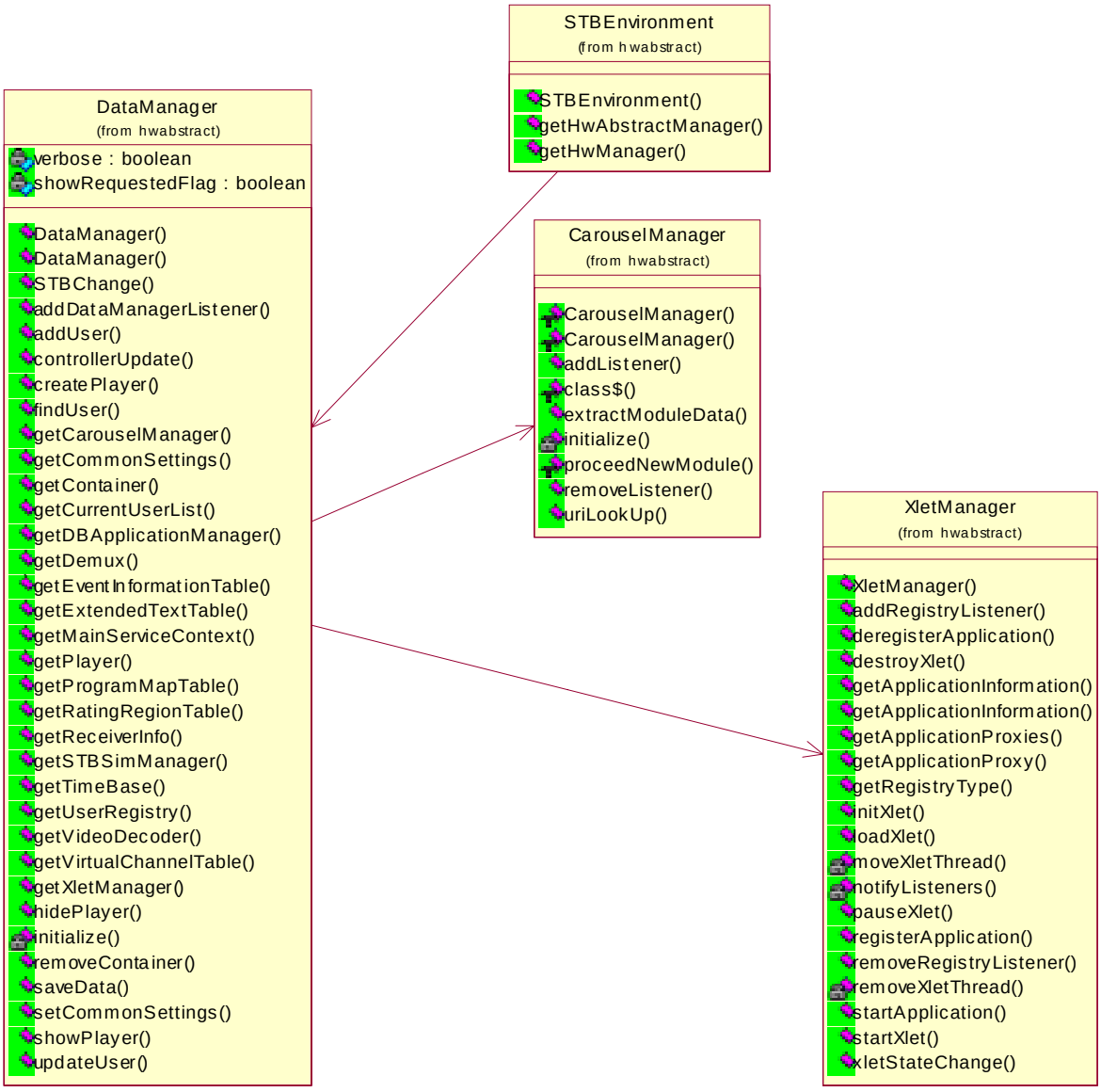


Figure 15 Main HAL Classes

9.1 Introduction

Figure 15 shows the main classes within the hardware abstraction layer. Even though classes outside of the HAL implement most of the API, a significant portion of the API is implemented by the HAL. For example, the Application Registry, the manager of DASE Applications, is implemented in the HAL. The reason for this is that there needs to be a single instance of the Application Registry, and it is therefore created in the HAL in order provide a consistent view of the registry to all Xlets. In most cases, API implementation objects do not exist unless an Xlet is running. However, in many cases the implementation

object needs to be created before any Xlet runs. Therefore, these objects are created and managed by the HAL `DataManager` object discussed below.

9.2 STB Environment

Class `STBEnvironment` manages information such STB state information, and User Profile information. This class communicates with the STB simulation in order to not only retrieve the STB data, but also to provide updates to the STB. An example of an update would be a new current User.

9.3 HAL Data Manager

The Hardware Abstraction Layer `DataManager` is the access point to all other HAL managers. It is also the access point for the API implementation to all data coming from the underlying STB implementation. This data includes ATSC PSIP information, and MPEG table data. Other STB information is accessed via the `STBEnvironment` class discussed in Section 9.2.

The `DataManager` listens for all `STBChangeEvents` (notification of changes to data maintained by the STB) and notifies any registered listeners of these changes. For example, the API implementation classes can register to be notified of changes in the User Registry

9.4 MPEG/PSIP Table Management

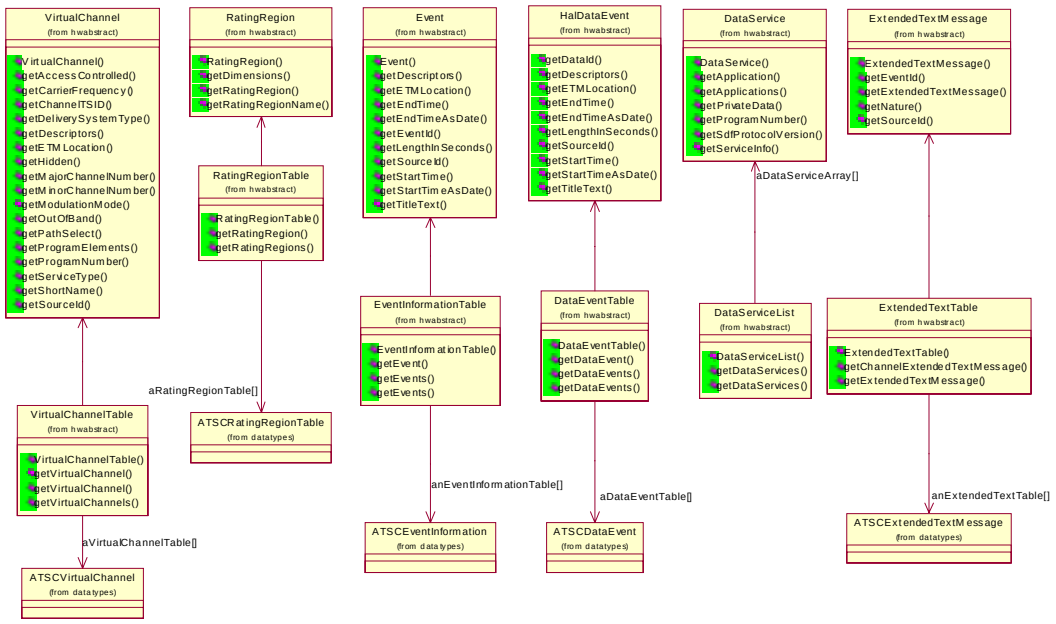


Figure 16 PSIP Tables in the HAL

9.4.1 Introduction

THIS SECTION NEEDS CLEANED UP…

Several classes with the HAL are used to provide access to the MPEG and PSIP table data accessed from the STB environment. These HAL classes isolate the API implementation from the format of the PSIP and MPRG table data as it is retrieved from the STB. Figure 16 shows the class diagram for the HAL classes that provide access to the PSIP data. Also shown in the diagram are the simulation datatype classes (those with a name beginning with ATSC).

9.4.2 Virtual Channels

Classes `VirtualChannel` and `VirtualChannelTable` are used to provide a standard interface to the API implementation classes to access virtual channel information. The `VirtualChannelTable` class encapsulates all of the functionality needed to map the virtual channel data format into a format needed by the implementation. This constructor of this class takes a reference to the simulation manager class. When porting takes place, the class will be rewritten to access the data in the manner native to the STB. Therefore, not only does this class map the virtual channel data, it is also responsible for encapsulating access to the STB.

Class `VirtualChannel` encapsulates the virtual channel data and provides accessor methods to retrieve the channel data.

9.4.3 Data Services

The classes used to by HAL to provide data service retrieval are `DataService`, `DataServiceList`, and `DataEventTable`.

The `DataEventTable` class provides an interface to the API implementation for Data Event information. All accesses to the underlying hardware/simulation layer is through `getXXX()` accessor methods. Overall the class provides the API with access to the simulation data in a convenient and portable way. The `DataEventTable` is a list of all the Data Events found in all the Virtual Channels (i.e., the HAL `VirtualChannelTable`).

The `DataService` class represents the information for a single Data Service as extracted from the Data Service Table. Class `DataServiceList` manages the list of `DataService` objects and provides consistent access to the list.

9.4.4 Event Information

HAL classes `Event` and `EventInformationTable` provide access to the Event Information Table (EIT). Class `Event` represents a single event take from the EIT. Retrieval of the event information is provided by accessor methods.

The `EventInformationTable` class provides an interface to the API implementation for Event information. All accesses to the underlying hardware/simulation layer is through `getXXX()` accessor methods. Overall the class provides the API with access to the STB data in a convenient and portable way.

9.4.5 Rating Information

There are several classes involved in providing Rating Region Table (RRT) information. These classes are `RatedDimension`, `RatedRegion`, `RatingRegion`, `RatingRegionTable`, and `LocalRatingRegion`. The `RatedDimension` class represents an instance of a rating dimension. The `RatedRegion` class represents an instance of a rating region. Class `RatingRegion` represents an instance of a rating region. A rating region may have multiple rating dimensions.

The `RatingRegionTable` class provides an interface to the API implementation for RRT information. All accesses to the underlying hardware/simulation layer is through `getXXX()` methods. Overall the class provides the API with access to the simulation data in a convenient and portable way. The class makes use of classes `RatingRegion` and `Dimension` that reflect the underlying stream structure of the

data as described in [ATSC:A65]. The information here represents data that is intended to be index into. For example, the RRT may contain a Dimension called MPAA that contains a rating table like, G, PG, PG-13, etc. Meta-data from rating descriptors can index into this data to obtain rating information about a Program or Service.

9.4.6 Descriptors

9.4.7 Extended Text Messages

The `ExtendedTextMessage` and `ExtendedTextTable` classes are used to access the Extended Text Table (ETT) information. Class `ExtendedTextMessage` represents a single entry of the Extended Text Table. Among others attributes, it holds the extended text message associated with a channel or event (EIT or DET).

The `ExtendedTextTable` class provides an interface to the API implementation for Extended Text Table (ETT) information. All accesses to the underlying hardware/simulation layer is through `getXXX()` methods. Overall the class provides the API with access to the simulation data in a convenient and portable way.

9.5 Data Broadcast

Figure 17 HAL Data Broadcast Classes

9.5.1 Description of Data Broadcast Classes in the HAL

The data broadcast standard is defined in [ATSC:A90]. Within the hardware abstraction layer, the broadcast data is managed in a manner that corresponds to the Data Carousel mechanism specified in [ATSC:A90]. The classes used to manage the Data Broadcast are shown in Figure 17.

9.5.1.1 CarouselManager

The HAL `CarouselManager` manages access to the data carousel resources downloaded by the underlying STB implementation. This class is designed to be instantiated and used by the HAL `DataManager` class. It implements URI lookup through Data Carousels in memory and module extraction based on a `CarouselFileLocator` (locator class internal to NIST implementation).

This class is only used to access Data Carousel modules that are associated with Carousel files or other data broadcast scenarios. This class does not manage the content associated with broadcast applications, such as Xlet class files. `CarousleManager` notifies the `CarouselFile` classes of changes in the Carousel file received by the STB environment.

9.5.1.2 CarouselModule

This class is a repository for data loaded out of the data carousel. It maps data from the underlying STB implementation into a form useful to the API implementation. This class stores all the metadata associated with the Carousel module, such as version, URI, etc.

9.5.1.3 DBDaseApplicationContent

This class is a data repository for storing information associated with a data broadcast application and related DASE bindings. It extends the `DBApplicationContent` class with DASE-specific information, such as Xlet class information.

9.5.1.4 DBApplicationContentFactory

This class creates `DBApplicationContent` objects from the information contained in the Data Service Table. There is also a method to create the `XletClassData` object for an Xlet by extracting the class byte codes from the STB environment.

9.5.1.5 DBApplicationContent

The parent class that is a data repository for data broadcast applications. It stores information specified by the data broadcast standard, such as application ID and program number.

9.5.1.6 DBApplicationManager

The HAL `DBApplicationManager` manages a list of applications in the ATSC Data Broadcast sense[1] as extracted from a Data Service Table. It gives read access to that list with a couple of methods (mainly used by the API implementation of the `javatv.service.selection` package). It also transmits notifications and, when needed, application resources to the `XletManager`. For example, when all application resources of a new application are received or when an application disappears from the DST.

[1] Defined in ATSC A/90 specification.

9.6 Application Management

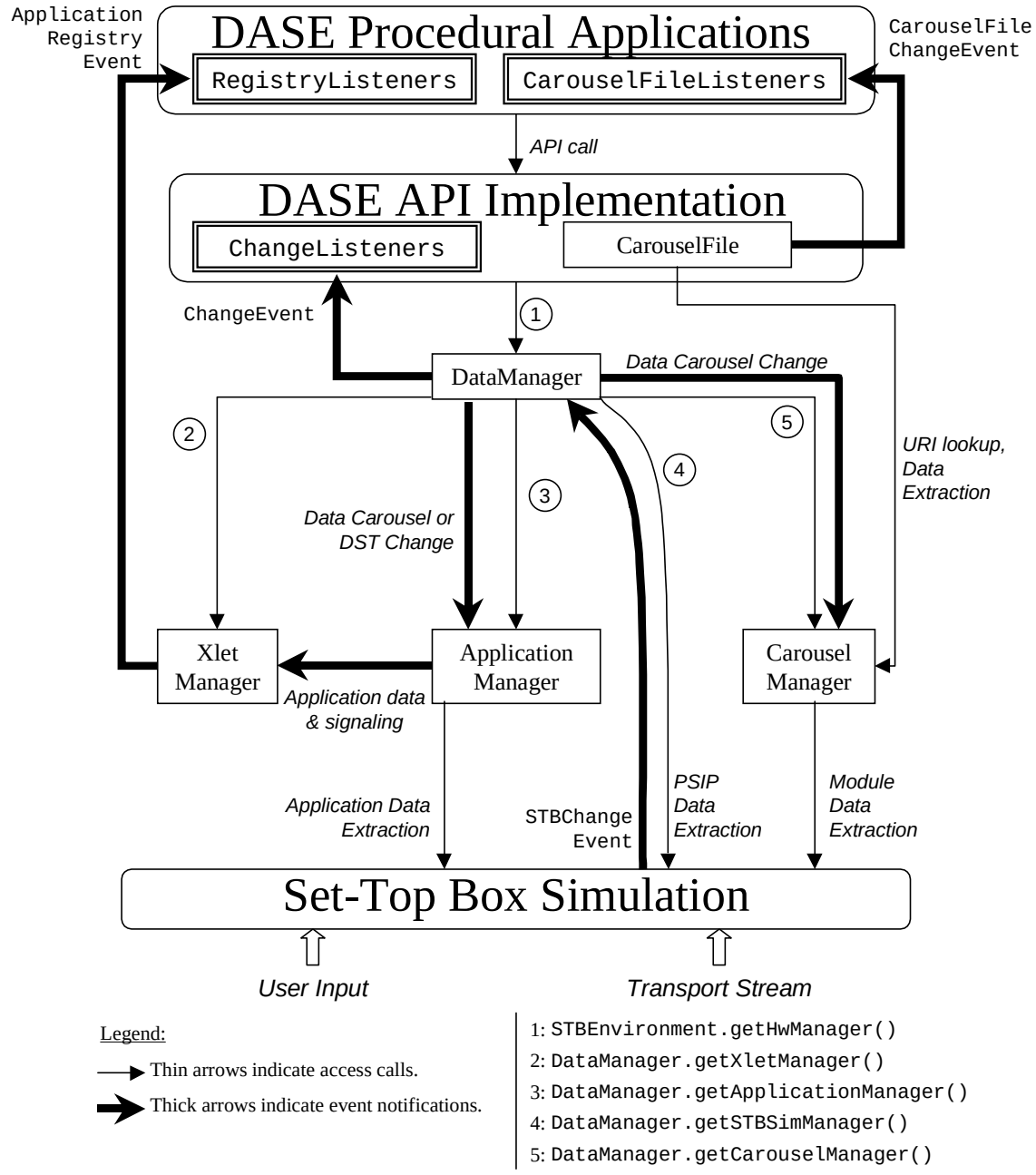


Figure 18 HAL Data Broadcast Interactions

9.6.1 Introduction

This section gives a more detailed description of all actions taken by both Simulation and HAL when receiving an application, from parsing the Transport Stream to launching Xlets and notifying API components. Figure 18 shows the interactions and relationships between the major components of the HAL involved in Data Broadcast application and Carousel management.

The context is a simple use case, actually the only one currently implemented by our Data Generator on one side, and our DASE PA platform on the other side. Basically, a new application is coming through the Transport Stream. Here are some characteristics:

- Everything is in one Transport Stream (no use of NRT).
- Data Carousels carry all application resources.

The steps involved in processing the DST are as follows:

1. A new DST is received and parsed by the Simulation, which in turns notifies the HAL. The HAL then examines the DST: if there's no new application (since previous DST in the same PMT Program), the process stops there.
2. If there is a new application, HAL checks if all resources are present by going through the Tap loop (arrows ③, ④, and ⑤ in Figure 18). For each Tap:
    2.1. Arrow ③: The HAL retrieves the PMT Program corresponding to this Data Service thanks to the `program_number` field in the DST. In that Program, the loop of elementary streams is then searched for an Association Tag Descriptor that matches with the `association_tag` of the Tap.
    2.2. Arrows ③ and ④: Using the `download_id` field of the Tap, the HAL locates the Data Carousel in the buffer. If the Tap points to an entire Data Carousel, HAL checks that Data Carousel has been completely received.
    2.3. Arrow ⑤: If the Tap points only to a single module, the HAL checks if that specific module has been received (Download Info Indication (DII) message and associated data).
3. If all resources indicated by the DST for the new application have been received, the HAL considers it as completely received and starts launching its entry points (Xlets).
4. In parallel to the treatment of the DST, Data Carousel modules are treated in a straightforward way. Each time a DII and its associated data are received, the Simulation considers the module as received and gives a notification to the HAL, which in turns notifies API-level objects listening for a change to that module (new module or new version of an existing module).

Also shown in Figure 18 is the Carousel Manager. When the HAL `DataManager` needs to access modules for a downloaded file (the data broadcast scenario) it relies in the `CarouselManager` to relay the module data from the STB environment.

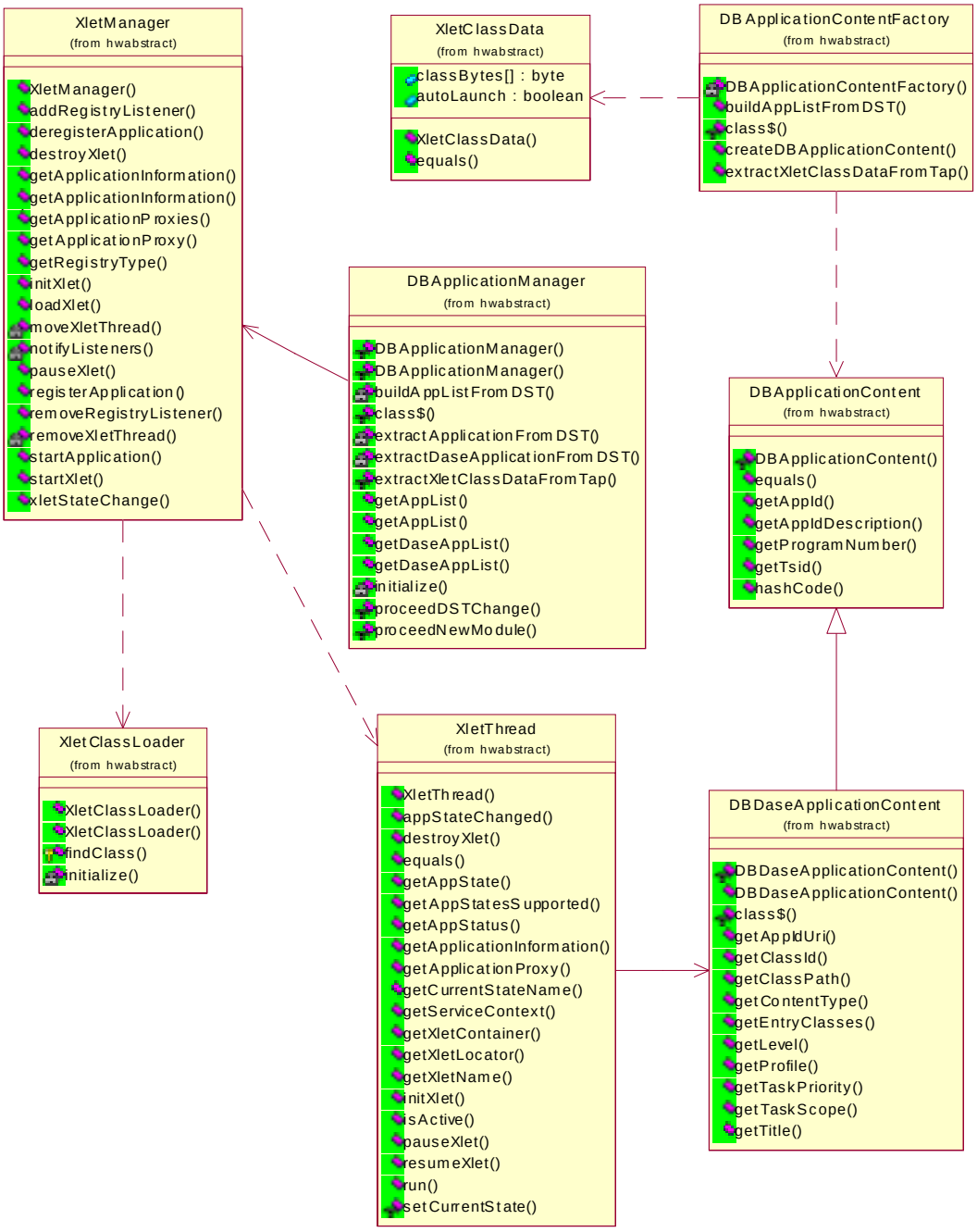


Figure 19 Class Diagram for Xlet Management

9.6.2 Xlet Management Classes

Figure 19 shows the classes involved in Xlet management. The following sections describe each class in more detail.

9.6.2.1 XletManager

The HAL `XletManager` manages Xlets (load, init, start, pause and destroy), based on the signals and aggregated data it receives from the `DBApplicationManager`. It implements `ort.atsc.ApplicationRegistry`, which includes managing and notifying a list of `org.atsc.RegistryListeners`. Xlets are controlled via the `XletThread` object that is created by the `XletManager`. The object centralizes the information for the Xlet, such as the Xlet's state, resource requirements, and other information.

9.6.2.2 XletThread

This class Centralizes information about the Xlet, such as its current state and status as maintained by the Xlet itself, the proxy objects, and the Locator for the Xlet. It is sub-classed from `Thread` and runs as a separate thread in the JVM.

9.6.2.3 XletClassLoader

This class implements the class loader for Xlets. The `XletManager` creates an object of this class for each Xlet. This class is a sub-class of `java.lang.ClassLoader`. The main responsibility of this class is to create the Java classes from the byte codes that were downloaded for the Xlet's classes. The delegation model is used where the parent class loader attempts to load the classes before calling on the Xlet class loader. Therefore, this class loader loads only classes that were downloaded by the system. The system class loader loads all other classes, such as the API implementation classes and P-Java classes.

Class `XletClassLoader` overrides method `findClass()` in order to load the Xlet classes. This method searches the private collection of byte codes downloaded for the Xlet. If the class is not found, a `ClassNotFound` exception object is thrown.

9.6.2.4 XletClassData

This class is a repository for the byte codes of a class that is part of an Xlet. It also stores information to indicate whether the class is the entry and launch point for an Xlet.

9.6.3 Xlet Resource Loading

Figure 19 shows the classes involved in Xlet loading and management. In the HAL, an Xlet is a form of Data Broadcast application. Therefore, loading of the Xlet's resources (class files, etc.) relies on the Data Broadcast classes introduced in Section 9.5.1. This section describes how these classes interact with the `XletManager` and `XletThread` classes.

Need more text here….

9.7 JMF Player

9.7.1 Abstract Decoder

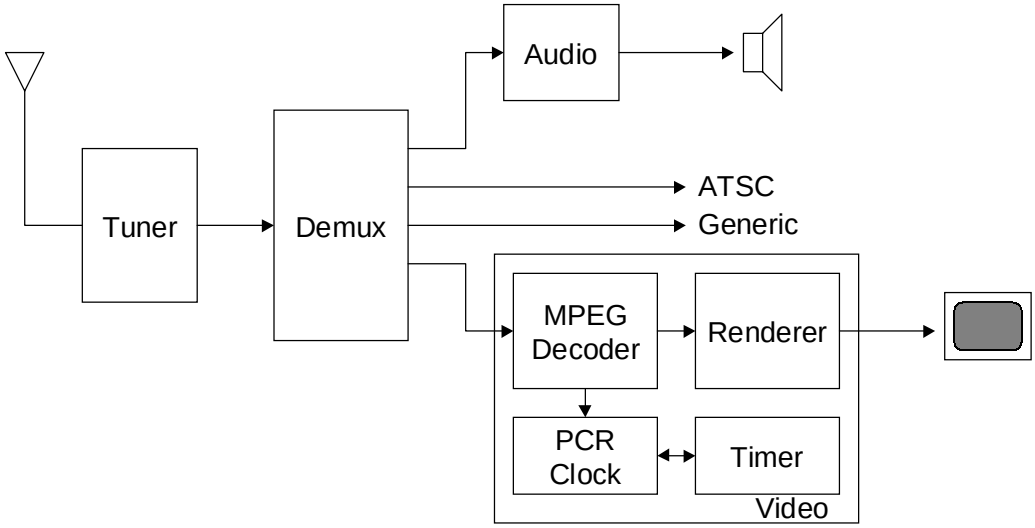


Figure 20 Set-Top Box Decoder Model

Figure 20 illustrates the general architecture of a set-top box hardware decoder. Although actual implementation choices may differ, a complete system should provide all the shown services, either in hardware or software (for example, many low-cost PC-based DTV decoders decode the AC-3 audio tracks in software).

In order to simplify porting of the Prototype Implementation to other platforms, we identified the following core services:

- Tuner: tuning services to select a specific transport stream.
- Demultiplexer (Demux): demultiplex the MPEG stream into elementary streams and route them to other blocks.
- Audio: audio decoding.
- Video:
    - Decoding of the MPEG video stream
    - Program Clock Reference: hardware clock phase-locked to the PCR timestamps within the video stream.
    - Timer: programmed alarm service associated with the clock.
    - Rendering.

This model identifies the lowest common denominator to low-level functions used by the Prototype Implementation, for convenience only. It is not intended to replace more extensive APIs such as JavaTV.

9.7.2 Java Interfaces

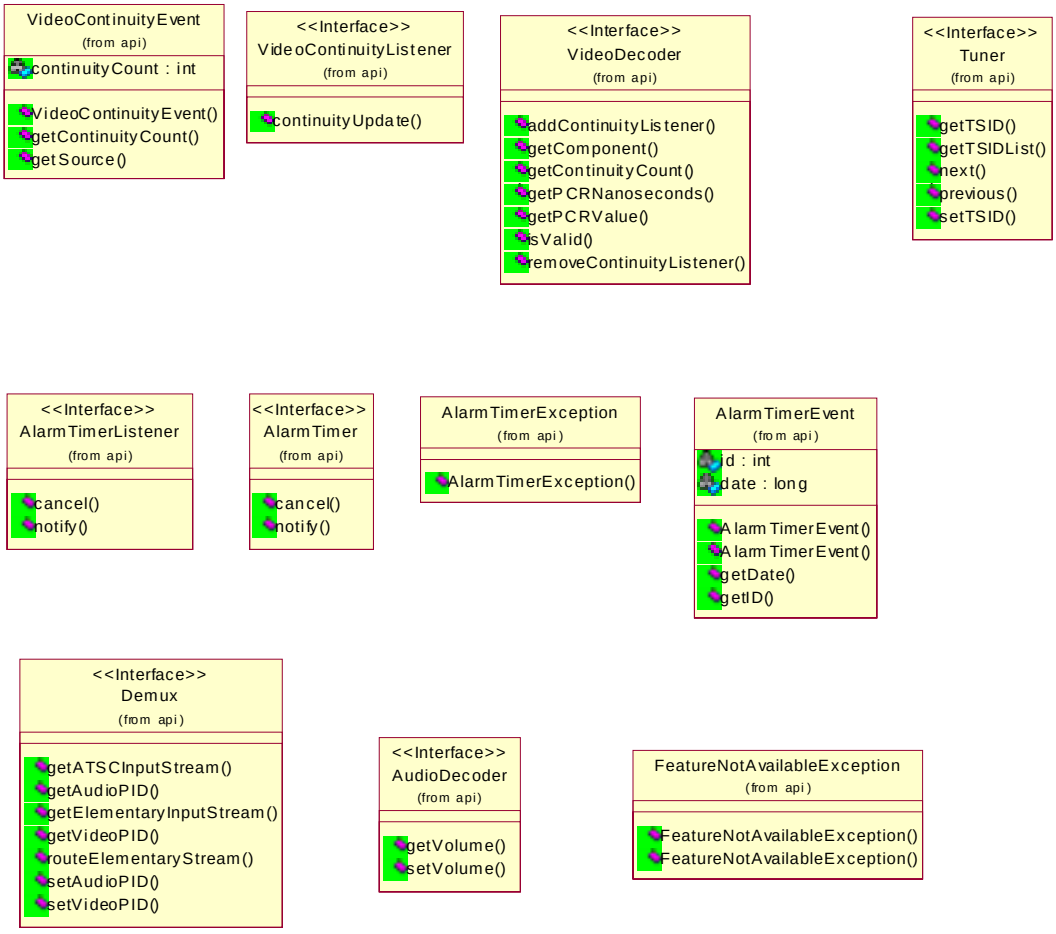
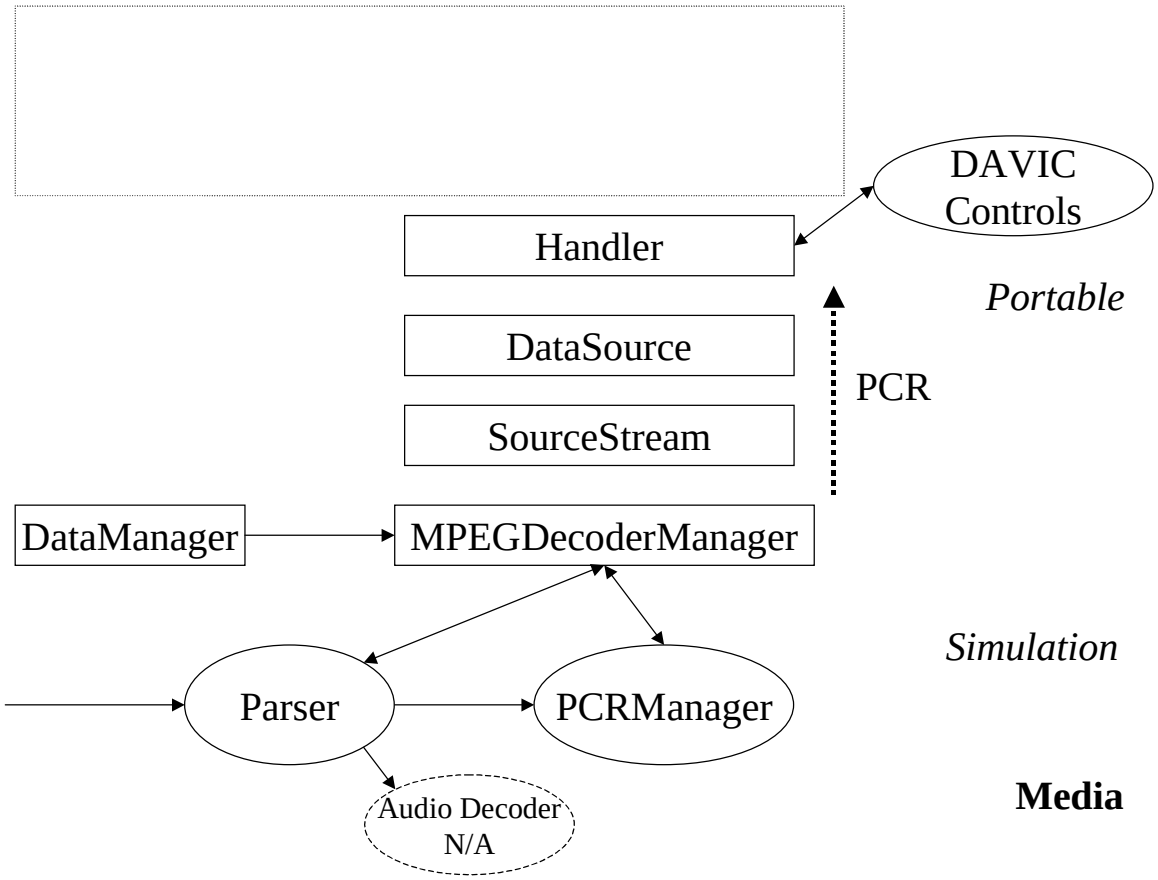


Figure 21 Java Interfaces for Decoder Components

The abstract model described in the previous section translates to a set of abstract hardware interfaces declared in `gov.nist.hwabstract.api`. Continuity (breaks in the PCR timeline) and Alarm events are handled via the usual Listener-Event mechanism. See the Javadocs for details**(NO, do it here –WJS).**


DAVIC
Controls
Handler
Portable
DataSource
PCR
SourceStream
DataManager
MPEGDecoderManager
Simulation
Parser
PCRManager
Media
Audio Decoder
N/A

10. API IMPLEMENTATION

10.1 Locators

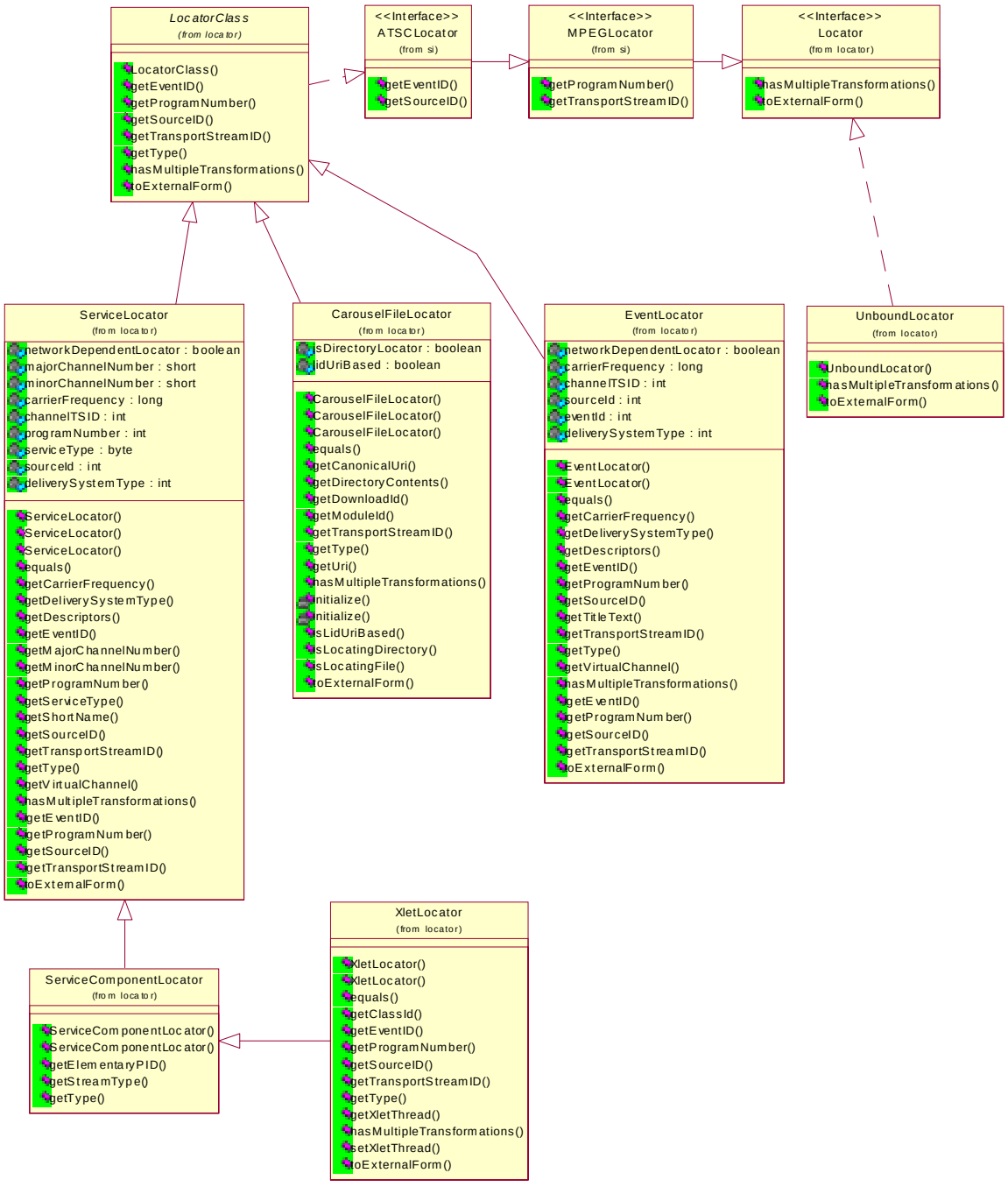


Figure 22 Locator Implementation

Locators are widely used throughout the JavaTV and DASE APIs for communicating implementation dependent information about a resource in an independent manner. Figure 22 shows the class diagram for the implementation of Locators in the DASE RI.

There are two forms of Locators in the implementation. One form is a Locator that contains information about a specific resource, such as Service or Xlet. This form of Locator is also referred to as a *bound* locator. The other form of Locator is *unbound* and does not contain any information about a resource, but may represent a resource not yet mapped into a specific stream component, for example. A Collection of bound Locators can be obtained from a unbound Locator in many instances.

10.2 The Management API

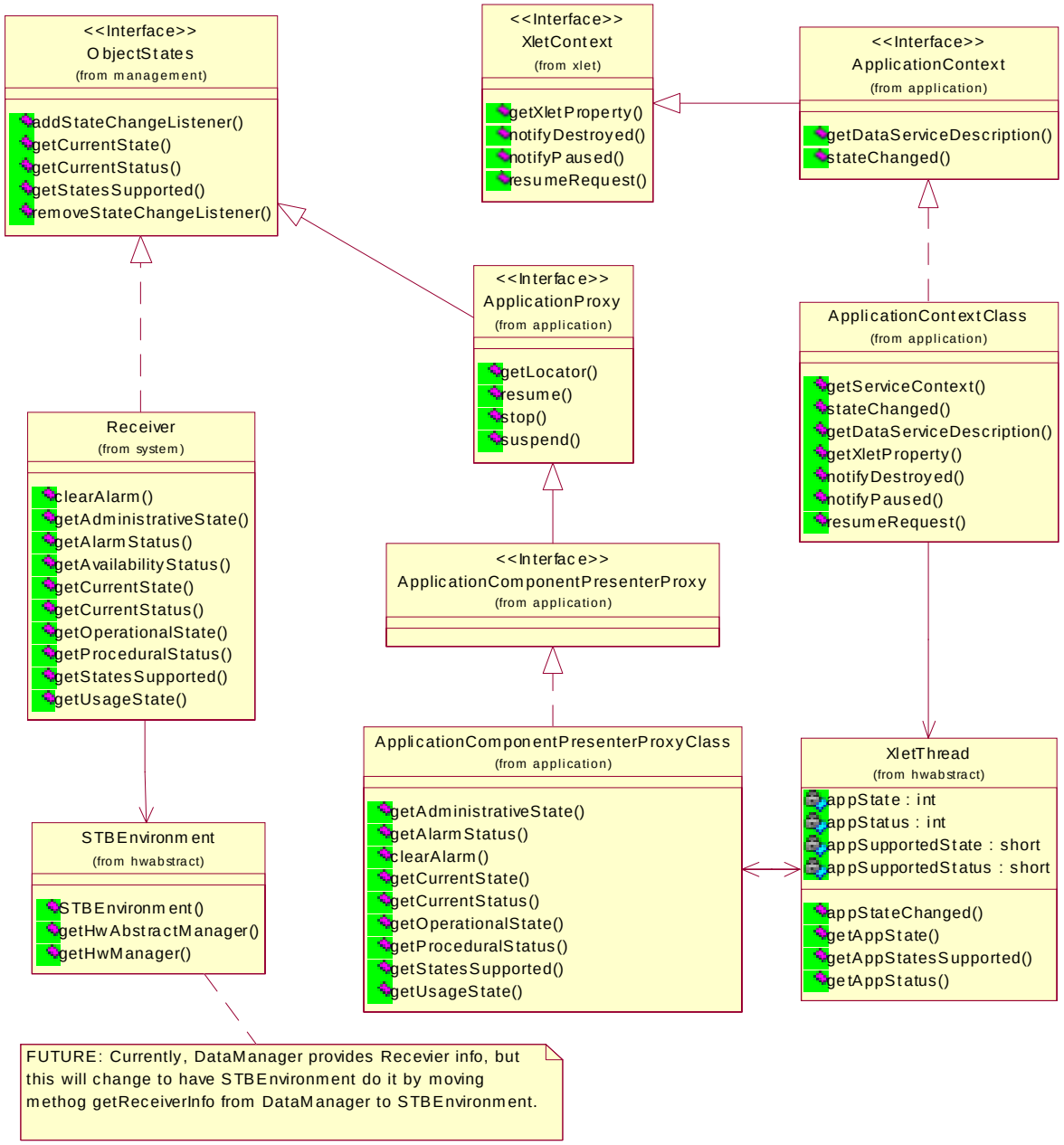


Figure 23 Management API Implementation

The Management API, specified in package `org.atsc.management`, is implemented by various packages inside the NIST RI. Also, Xlets themselves can support the Management states and statuses via the `ApplicationContext` object. The values of the states and statuses reported by the Xlet are maintained in the `XletThread` object associated with the Xlet. Section 9.6.2.2 discusses the `XletThread`

class. Figure 23 shows how the state and status information for an Xlet is maintained within the API implementation and how that information is retrieved via the `ApplicationProxy` interface.

Several classes in the NIST RI provide support to classes that implement the `ObjectStates` interface in order to provide a minimal level of support for Management as suggested by the DASE PAE standard. For example, HAL class `STBEnvironment` maps information from the underlying STB environment into Management states and statuses. The class `org.atsc.system.Receiver`, which implements interface `ObjectStates`, retrieves this information in order to support the `ObjectStates` interface. The actual mapping of STB states and statuses takes place in the `STBEnvironment`.

10.3 Service APIs

10.3.1 Overview

The Service API package gives the Xlet access to the Service Information (SI) database and the mechanism to tune to (select) individual Services. The parent javax.tv.service package defines what a Service is and provides the access class SIManager to discover the available Services. Sub-packages of javax.tv.service include navigation that allows the Xlet to navigate the collection of Services. The guide package provides detailed information about the Services suitable for an Electronic Program Guide. Service selection provides the mechanisms to tune to and start selected Services. The transport package isolates some of the specific delivery media information (currently MPEG-2).

10.3.2 Asynchronous Service Information Retrieval

The Service package provides mechanisms to retrieve data from the SI database asynchronously. Methods that begin with the retrieve prefix provide this asynchronous data retrieval. The NIST implementation handles asynchronous methods by creating a class that is a separate thread. The initial call to the retrieve method will create and start the new thread. The call will then return immediately. The asynchronous thread will retrieve the data from the SI database and will notify listeners of the outcome. **(Provide generic code for the general solution).**

10.3.3 Package javax.tv.service

Specification	Implementation Mapping	Level
Interfaces		
DeliverySystemDef		
Service	gov.nist.service.ServiceClass	
ServiceComponent	gov.nist.service.ServiceComponentClass	
ServiceMinorNumber	extended by javax.tv.Service	
ServiceNumber	extended by javax.tv.Service	
SIChangeListener		
SIElement	extended by many interfaces	
SIRequest		
SIRequestor		
SIRetrievable		
Classes		
DeliverySystemType		
ReadPermission		
ServiceInformationType		
ServiceType		
SIChangeEvent		
SIChangeType		

SIRequestFailureType		
StreamType		
Exceptions		
SIException		

Table 4 Service Implementation Mapping

10.3.4 Package javax.tv.service.guide

10.3.4.1 Guide UML Diagram

The root for obtaining guide information is the ServiceDetails object. The ServiceDetails object will contain a ProgramSchedule that provides the mechanisms for retrieving ProgramEvents. ProgramEvents are retrieve asynchronously. A Xlet will make a call to the ProgramSchedule object to obtain certain events. This call will return immediately with a SIRequest object. The SIRequest object is used to later retrieve the desired information. ProgramSchedule relies on a set of asynchronous retrieve classes to handle the data retrieval. See section X.X on the implementation of asynchronous methods.

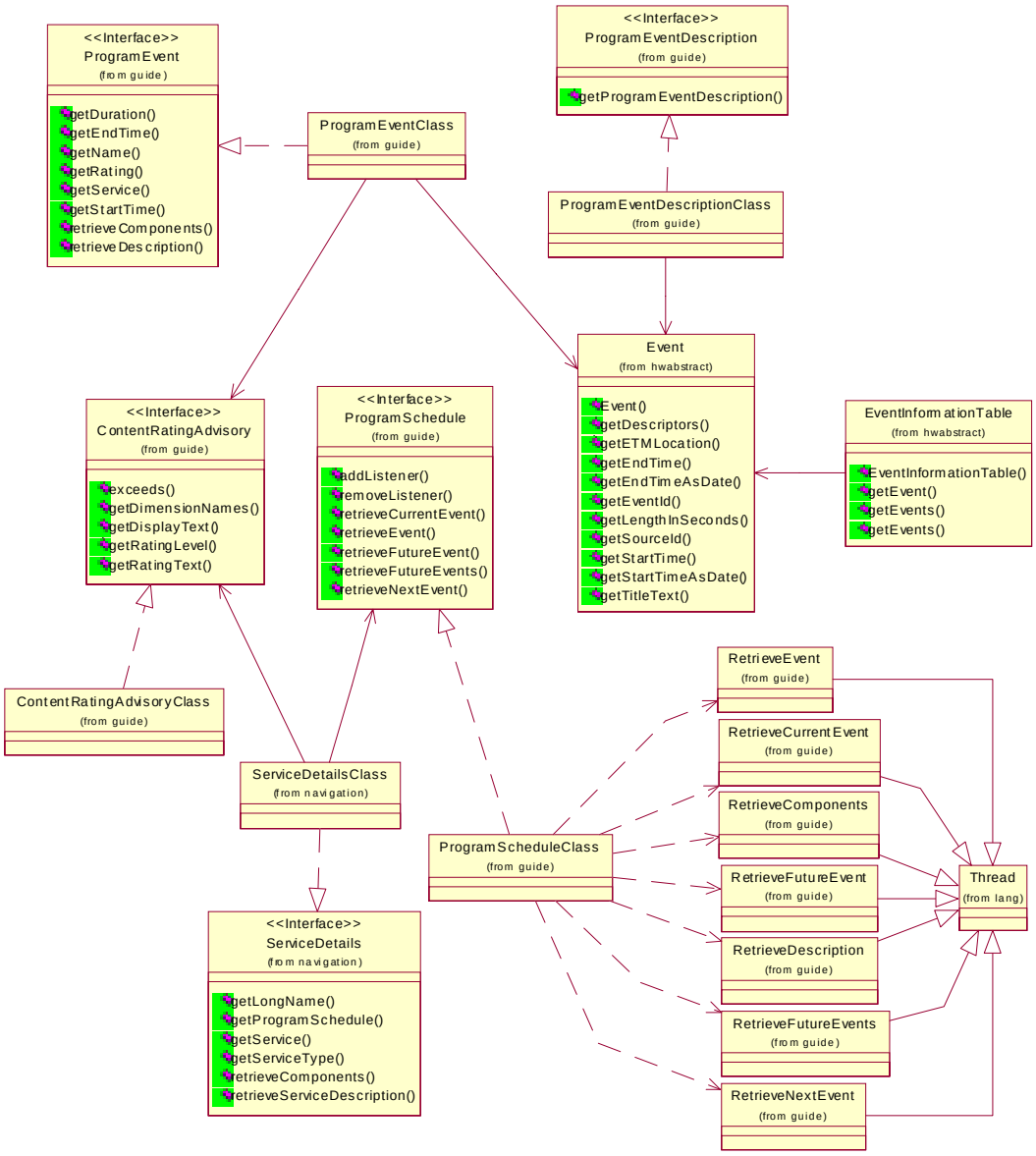


Figure 24 Guide UML Diagram

The data used to implement Program Events is obtained from the HAL EventInformationTable and Event classes. The EventInformationTable (EIT) contains the aggregate of Events across Virtual Channels. Events can be retrieve from the EIT via the source_id, which will return all the Events for a given virtual channel. Events can also be retrieved individually with a source_id and event_id.

10.3.4.2 Specification to Implementation Mapping

Specification	Implementation Mapping	Level
Interfaces		
ContentRatingAdvisory	gov.nist.service.guide.ContentRatingAdvisoryClass	
ProgramEvent	gov.nist.service.guide.ProgramEventClass	
ProgramEventDescription	gov.nist.service.guide.PrgoramEventDescriptionClass	
ProgramSchedule	gov.nist.service.guide.ProgramScheduleClass	
ProgramScheduleListener	Place holder for new version of specification	
Classes		
ProgramScheduleChangeEvent		
ProgramScheduleChangeType	Place holder for new version of specification	
ProgramScheduleEvent	Place holder for new version of specification	

Table 5 Guide Implementation Mapping

10.3.4.3 Implementation to ATSC/MPEG Table Mapping

API Object	Attribute	Implementation	Simulation	ATSC
ProgramEvent	name	gov.nist.service.guide.ProgramEventClass, gov.nist.hwabstract.EventInformationTable, gov.nist.hwabstract.Event		EIT
ProgramEvent	duration			EIT
ProgramEvent	startTime			EIT
ProgramEvent	endTime			EIT
ProgramEvent	contentAdvisory			content_advistory_descriptor
ProgramEvent	description			EIT
ProgramEvent	components			VCT (service location descriptor)
RatingDimension	name			RRT
RatingDimension	# of Levels			RRT
RatingDimension	levelDescription			RRT
ContentRatingAdvisory	dimensionName	ContentRatingAdvisoryClass		content_advisory_descriptor
ContentRatingAdvisory	ratingValue			content_advisory_descriptor
ContentRatingAdvisory	ratingText			content_advisory_descriptor
ContentRatingAdvisory	displayText			content_advisory_descriptor

Table 6 Guide Implementation to Transport Mapping

10.3.5 Package javax.tv.service.navigation

10.3.5.1 Overview

The navigation package provides the mechanisms to conveniently browse through the available Services list. It arranges Services as collections and provides information about the Services.

10.3.5.2 Specification to Implementation Mapping

Specification	Implementation Mapping	Level
Interfaces		
CAIdentification		
FavoriteServicesName		
RatingDimension	gov.nist.service.navigation.RatingDimensionClass	
ServiceCollection	gov.nist.service.navigation.ServiceCollectionClass	
ServiceDescription	??	
ServiceDetails	gov.nist.service.navigation.ServiceDetailsClass	
ServiceIterator	gov.nist.service.navigation.ServiceIteratorClass	
ServiceProviderInformation	optionally implemented by ServiceDetails. Not implemented in the NIST implmentation.	
Classes		
LocatorFilter	gov.nist.service.navigation.LocatorFilter	
PreferenceFilter	gov.nist.service.navigation.PreferenceFilter	
ServiceFilter	gov.nist.service.navigation.ServiceFilter	
ServiceTypeFilter	gov.nist.service.navigation.ServiceTypeFilter	
SIElementFilter	gov.nist.service.navigation.SIElementFilter	
SIManager	gov.nist.service.navigation.SIManagerClass, supported by RetrieveServiceDetails, RetriveService, RetrieveEvent	
Exceptions		
FilterNotSupportedException		
NoSuchServiceException		
SortNotAvailableException		

Table 7 Navigation Implementation Mapping

10.3.6 Package javax.tv.service.selection

10.3.6.1 Overview

The Service Selection API allows applications the ability to control the presentation of Services in a simple high-level way. It allows the presentation of a Service without the application having to know the details of the Service.

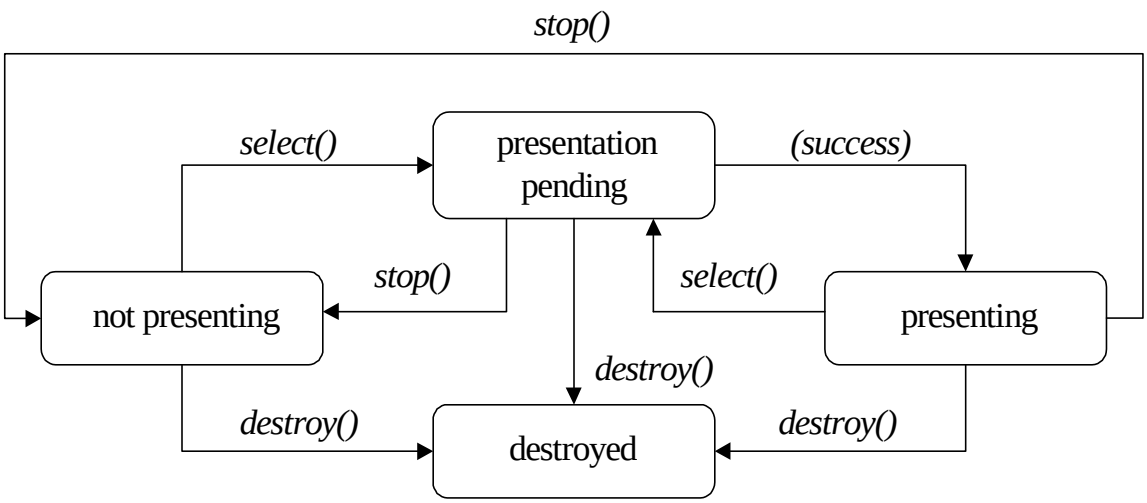


Figure 25 Service Context State Diagram

The application requesting a Service Selection first creates a `ServiceContext` object that will control the selection and presentation of Services. Through the `ServiceContext` the application can select Services and Service Components with the use of the Locators. Once the `select()` method is called the component locators are discovered and the appropriate component presenters are instantiated to present the components.

The Service Selection package gives a DASE application the ability to select a Service or a component within a Service for presentation. If a Service is selected, then all components of that Service will attempt to be presented. Services are presented and managed in a Service Context. There can be multiple Service Contexts in a DASE receiver; however, the NIST Set-top box limits this to one Service Context, which can be created by the DASE application or the implementation. The initial Service Context is created by the implementation. This implies that when a DASE application begins to execute it can obtain a reference to the implementation Service Context for controlling the presentation of Services.

A Service or Service Component is selected by first obtaining a reference to a Service Context object. The DASE application can create a Service Context object via the Service Context Factory or by obtaining a reference to the implementation Service Context. Using the Service Context the DASE application can call the select method with a Service Locator or a Service Component Locator.

The Service Context Class manages the presentation of Services. The select and stop methods control the starting and termination of Services. The Service Context uses the Service Context State object to maintain its state. Table [Selection State] describes the possible states of a Service Context. Once a Service or Service Component has been selected, the task of presenting it is handed off to the Service Component Presenter class. The call to select returns to the DASE application, Service Component Presenter is an asynchronous class.

The Service Component Presenter class first determines the intent of the DASE application by examining the locators and the state of the Service Context. If the Service Context is in the NOT_PRESENTING State, then SCP attempts to present the Service Components represented by the Locators. SCP determines the resources needed and attempts to retain them. For example, if one of the Locators map to an elementary video stream, then access to the Player is requested.

`ServiceComponentPresenter` is a helper class for the `select()` method in `ServiceContext`. This is an asynchronous class that initiates the presentation of a Service or individual `ServiceComponents`. Based on the locator(s), `ServiceComponentPresenter` selects the appropriate "engine" to present the `ServiceComponent`.

`ServiceComponentPresenter` is an asynchronous class that determines the selected Service Components and plays them in the appropriate Handler, whether it is a JMF Player or Xlet presenter.

10.3.6.2 Access to the JMF Player

The Service Component Presenter will often need to obtain access to the JMF Player in order to select components mapping to video and audio streams. Access to the Player is obtained in the HAL `DataManager`.

The `DataManager` creates a Player (i.e., `ServiceMediaHandler`) object. It does this by calling the `createPlayer()` method in the `java.media.Manager` class. The `createPlayer()` method takes a `MediaResourceLocator` (MRL) object that is a URL that points to the actual system (i.e., real hardware) player. `Manager.createPlayer()` searches the Locator path looking for a player implementation that matches the one indicated by the URL. In the case of the NIST RI, it is the Handler implementation class. So the Player in `DataManager` is really an instantiation of the Handler class in the HAL. Handler implements `ServiceMediaHandler`, so the Player is a `ServiceContentHandler`. A `ServiceContentHandler` object is accessible by the Xlet for manipulation of the JMF Player.

10.3.6.3 Policy and Issues

Future: Complete this table

Current State	Action	Outcome	Result	New State	Comments
Presenting	select()	Success	NormalContentEvent	Presentation Pending	
Presenting	select()	Success	AlternativeContentEvent	Presenting	
Presenting	select()	Failure	SelectionFailedEvent	Presenting	
Not Presenting	select()	Success	NormalContentEvent	Presentation Pending	
Not Presenting	select()	Success	AlternativeContentEvent	Presenting	
Not Presenting	select()	Failure	SelectionFailedEvent	Presenting	
Destroyed	select()	None	IlegalStateException	Destroyed	

Table 8 Service Context State Table

If the implementation restricts the number of ServiceContext objects to one and an application that is already running:

Scenario: Another application tries to create a `ServiceContext` object

Solution 1: the `ServiceContext` creation fails

Solution 2: may succeed to get the `ServiceContext` object that is currently used by the previous application. Then the `ServiceContext` object sends such as `PresentationTerminatedEvent` to the previous application. Previous application should be terminated by itself.

Question: Should the new application kill the previous application? Is this the responsibility of the Application Manager?

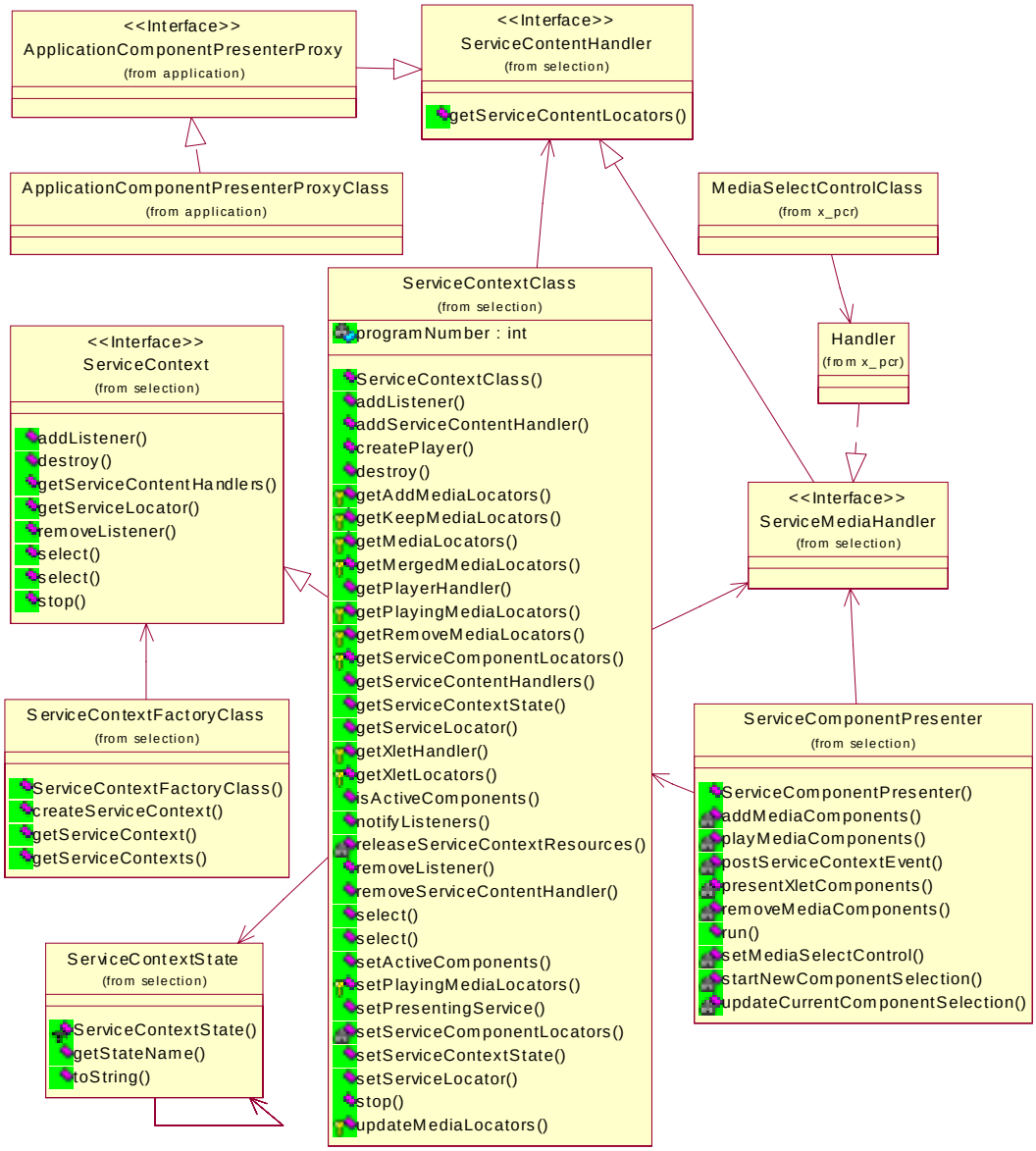


Figure 26 Service Selection UML Diagram

10.3.6.4 Service Selection UML Diagrams

The Service Selection implementation classes are shown in Figure 26.

10.3.6.5 Specification to Implementation Mapping

Specification	Implementation Mapping	Level
Interfaces		
ServiceContentHandler	gov.nist.hwabstract.	
ServiceContext	gov.nist.service.ServiceContextClass	
ServiceContextListener	Implemented by DASE Application	
ServiceMediaHandler	gov.nist.hwabstract.	
Classes		
AlternativeContentEvent		
NormalContentEvent		
PresentationChangedEvent		
PresentationTerminatedEvent		
SelectionFailedEvent		
SelectPermission		
ServiceContextDestroyedEvent		
ServiceContextEvent		
ServiceContextFactory		
ServiceContextPermission		
Exceptions		
InsufficientResourcesException		
InvalidServiceComponentexception		
ServiceContextException		

Table 9 Service Selection Implemetnation Mapping

10.3.6.6 Implementation to ATSC/MPEG Table Mapping

API	Implementation Mapping	Tables
ServiceComponent	gov.nist.hwabstract.	
ServiceContext	gov.nist.service.ServiceContextClass	
ServiceContextListener	Implemented by DASE Application	
ServiceMediaHandler	gov.nist.hwabstract.	
AlternativeContentEvent		
NormalContentEvent		
PresentationChangedEvent		
PresentationTerminatedEvent		

Table 10 Service Selection Implementation to Transport Mapping

10.3.6.7 Notes and Issues

10.3.7 Package javax.tv.service.transport

10.3.7.1 Overview

10.3.7.2 Specification to Implementation Mapping

Specification	Implementation Mapping	Level
Interfaces		

Bouquet	Not required for ATSC implementation	
BouquetCollection	Not required for ATSC implementation	
Network	Not required for ATSC implementation	
NetworkCollection	Not required for ATSC implementation	
Transport		
TransportStream		
TransportStreamCollection		
Classes		
BouquetCollectionChangeEvent	Not required for ATSC implementation	
NetworkCollectionChangeEvent	Not required for ATSC implementation	
TransportStreamCollectionChangeEvent		

Table 11 Transport Implementation Mapping

10.3.7.3 Notes and Issues

`javax.tv.service.transport.Network` and `NetworkCollection` are not required for ATSC implementations. `javax.tv.service.transport.Bouquet` and `BouquetCollection` are not required are not required for ATSC implementations. These classes are part of a generic JavaTv specification that is used in other specification, such as DVB.

10.4 User and Preference Management

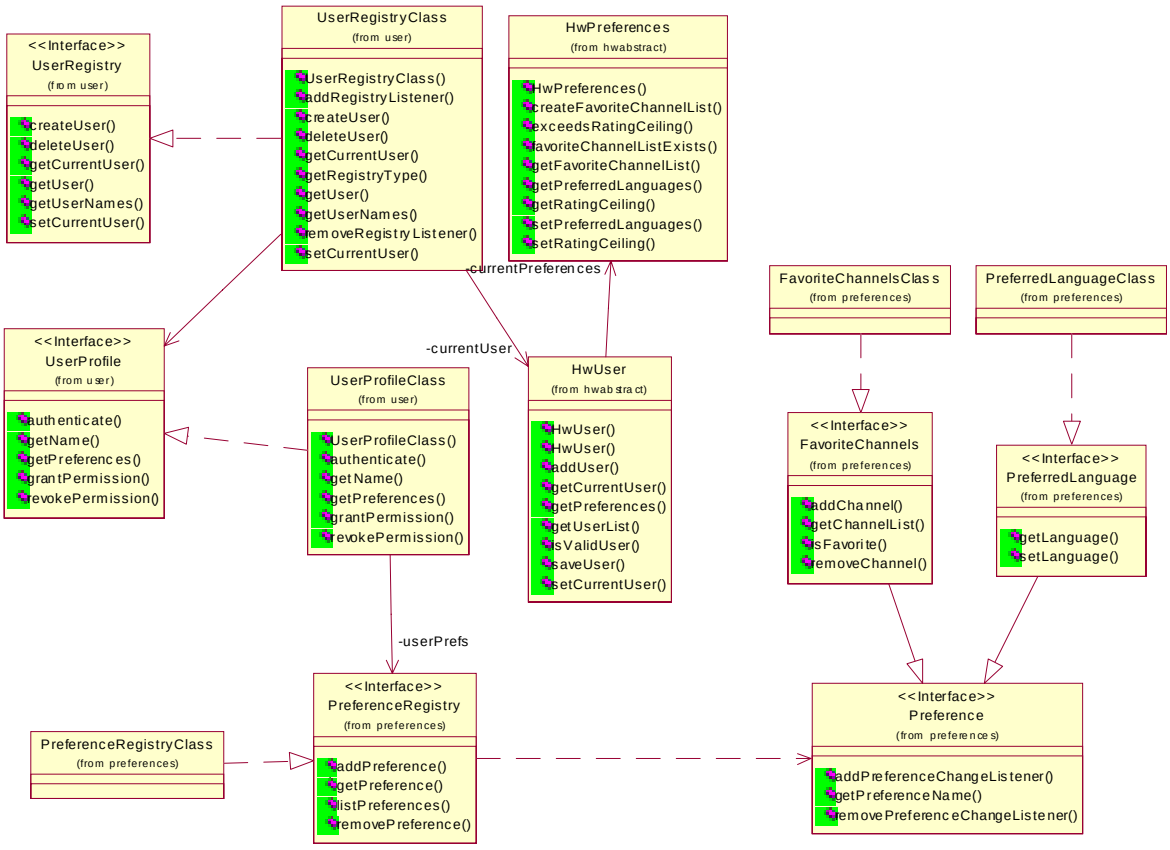


Figure 27 Class Diagram for User Management

The DASE API defines several interfaces and classes that comprise the Xlet's view of STB users and their associated preferences. Figure 27 shows some of the interfaces and implementation classes associated with User management. User accounts are stored in a Registry whose interface is defined by the DASE API. In the NIST RI there exists one and only one copy of the User Registry in the JVM. The accounts themselves are referred to as User Profiles in the DASE specification. Associated with each User Profile is another Registry that is used to store individual User Preferences. There are four types of User Preferences: Favorite Channels, Preferred Language, Rating Preference and Personal Data.

A Favorite Channels preference is essentially a list of preferred channels that has a name associated with it (e.g. "Sports Channels"), and there can exist any number of them in a user's Preference Registry. In contrast, the latter 3 Preference types may exist only as a single instance per Preference Registry. Preferred Language is an ordered list of language codes, sorted in order of preference. Ratings Preference is a set of "Rating Dimension" and "Rating Value" ordered pairs. Personal Data is a set of Attribute and Value ordered pairs that are wholly user defined.

Note that the underlying STB may or may not support multiple users with individual preferences. In either situation at least one "user" account is assumed to exist, and is referred to as the *common user* or *common settings*. The NIST STB simulation environment in fact provides for multiple users, however the

NIST DASE RI code (via the Hardware Abstraction Layer) is designed to function correctly on a STB that only supports the single user scenario.

Specification	Implementation Mapping	Level
Interfaces		
UserRegistry	gov.nist.user.UserRegistryClass	
UserProfile	gov.nist.user.UserProfileClass	
UserPermissions	org.atsc.user.UserPermissions	
FavoriteChannels	gov.nist.preferences.FavoriteChannelsList	
PersonalData		6
Preference	Implemented by classes that implement sub-interfaces	1
PreferenceChangeListener	Implemented by Xlets	1
PreferenceNames	No implementation needed	
PreferenceRegistry	gov.nist.preferences.PreferenceRegistryClass	
PreferredLanguage	gov.nist.preferences.PreferredLanguageClass	
RatingPreference	gov.nist.preferences.RatingPreferenceClass	
Classes		
UserChangeCause	org.atsc.user.UserChangeCause	1
UserPermission	org.atsc.user.UserPermission	4
UserRegistryEvent	org.atsc.user.UserRegistryEvent	1
LanguageScope	org.atsc.preferences.LanguageScope	1
PreferenceChangeCause	org.atsc.preferences.PreferenceChangeCause	1
PreferenceChangeEvent	org.atsc.preferences.PreferenceChangeEvent	1
PreferencePermission	org.atsc.preferences.PreferencePermission	4
PreferenceRegistryEvent	org.atsc.preferences.PreferenceRegistryEvent	1
Exceptions		
InvalidPermissionException	org.atsc.user.InvalidPermissionException	1
InvalidUserException	org.atsc.user.InvalidUserException	1
InvalidPreferenceException	org.atsc.preferences.InvalidPreferenceException	1

Table 12 User and Preference Implementation Mapping

The implementation of User and Preferences management is spread over several packages, namely: `org.atsc.registry`, `org.atsc.user`, `org.atsc.preferences`, `gov.nist.preferences`, and `gov.nist.hwabstract`. In the sections that follow, the details of the implementation will be discussed.

10.4.1 User Registry

Upon startup of the NIST STB simulation environment, an object of class `gov.nist.hwabstract.DataManager` is instantiated. The NIST RI maintains a single copy of the `DataManager` object for use by all Xlets. Xlets that wish to access the User Registry do so by (as defined in the DASE specification) using the `RegistryFactory`. The `RegistryFactory` contains a method to return a reference to the User, Application, Preference or Resource Registry. It should be noted that the Resource Registry is not specified by the current version of DASE, and the Resource registry type is currently a placeholder for future use. Note also that the `PreferenceRegistry` returned by the Registry Factory contains the Preferences belonging to the “common user” or “common settings” only.

The UserRegistry object retrieved using the RegistryFactory is obtained from the DataManager. The first time it is retrieved (after the STB Simulation is started) the UserRegistry object is instantiated and populated with User and Preference records for each STB user. Thereafter, calls to the Registry Factory to retrieve the UserRegistry will return a reference to this same single instance.

The UserRegistry maintains a table of UserProfile objects for each STB user. Each UserProfile in turn points to single PreferenceRegistry that contains the user's Preferences. Changes made to a UserProfile in the UserRegistry, their associated Preference Registries, or individual Preferences contained within those preference registries will result in immediate changes to the User and Preference information contained in the STB. This synchronization of information between the UserRegistry and the STB is achieved via classes defined in the HAL that are single threaded. (Note that this isn't currently true of the HAL classes, only of the STB ones so there may in fact be a potential for race condition problems) Thus, all Xlets view User and Preference information in a consistent manner, because all changes to the underlying STB information are synchronized and confined to a single copy in the STB.

In addition to the storage of User Profile information, the UserRegistry maintains some state information. The UserRegistry maintains the current login state as the name of the currently logged in user. The UserRegistry interface provides methods that allow the currently logged in user to be retrieved or set. In the latter case some sort of authentication procedure would normally be performed, however the NIST RI currently provides no security mechanisms. Thus, any Xlet can reset the currently logged in user state of the UserRegistry at any time.

The UserRegistry interface (as specified by DASE) provides a mechanism for asynchronous notification of Xlets when changes occur to the UserRegistry, namely when UserProfile objects are added or deleted. A current limitation of the NIST RI is that notification of Xlets occurs in a single thread, and thus Xlets that fail to respond can block other Xlets from receiving notification events.

10.4.2 User and Preferences Classes in the HAL

The HAL classes HwUser and HwPreferences provide the API implementation classes with a standardized interface to the STB User and Preference information. In order to populate the UserRegistry, a single HwUser is instantiated (using the no argument constructor) in order to gain access to the list of STB usernames. Once this list has been obtained, an HwUser object is instantiated for each STB user, using each username as an argument to the HwUser constructor.

When HwUser is instantiated, it automatically instantiates a HwPreferences object, to which it maintains a reference. The HwPreference object, when instantiated, loads all of the STB preference information for the specified user.

In addition to providing methods to create/delete/modify user and preference information, Class HwUser provides several utility methods that are used by the UserRegistry class:

```
setCurrentUser()
getCurrentUser()
getUserList()
isValidUser()
getPreferences()
```

It is important to note that each instance of HwUser and HwPreferences contains a copy of the associated STB User and Preference information. Thus, only one copy of each should be instantiated per-user otherwise inconsistencies may result.

HwPreference objects encapsulate all User Preference information stored in the STB. These objects are only created when HwUser is instantiated, and maintain a reference back to the HwUser object that created them. HwPreferences provides methods to create/delete/modify/retrieve favorite services lists, the preferred language list, or the set of rating ceilings stored in the STB. In addition, class HwPreferences provides several utility methods that are used by other API classes implementing User Preferences:

```
exceedsRatingCeiling()
setRatingCeiling()
getRatingCeiling()
setPreferredLanguage
favoriteChannelListExists()
getFavoriteChannelList()
createFavoriteChannelList()
```

10.4.3 Preference Registry and Preference

During creation of the UserRegistry object, the HAL routines build a list of HwUser (and their associated HwPreferences) objects for each STB user, along with a PreferenceRegistry object for each user.. Utilizing these objects, a Preference interface compliant object is then instantiated for each individual preference, depending upon its type, added to the user's PreferenceRegistry. For example, if a given user has three Favorite Services lists at the STB level – as can be determined using HwPreferences – then three objects are created of type FavoriteChannelsClass and added to the user's PreferenceRegistry.

10.4.3.1 User Profile

After all PreferenceRegistry objects have been created and populated for each STB user, a UserProfile object is then created for each user using the list of PreferenceRegistry objects in the constructor. UserProfile objects are then able to return a reference to the user's PreferenceRegistry upon demand. Once all UserProfile objects have been instantiated for each user, the UserRegistry itself is then instantiated using the list of UserProfile objects in the constructor.

10.4.3.2 Event Handling

Events are fired at three separate levels throughout the User and Preferences implementation (as specified by the DASE API). Events are generated by the UserRegistry, and also by the individual Preference Registries, and the Preference objects themselves. Xlets can register as listeners at any of the three levels as desired.

10.4.3.3 User Registry Events

`UserRegistryEvent` objects are fired under these conditions:

1. A new current User has been set
2. A new User has been added
3. A User has been deleted

Note that event notification is currently implemented in a single thread, and thus there is a potential for problems if Xlets fail to respond.

10.4.3.4 Preference Registry Events

`PreferenceRegistryEvent` objects are fired under these conditions:

- A Preferences has been added
- A Preferences has been removed

Note that event notification is currently implemented in a single thread, and thus there is a potential for problems if Xlets fail to respond

10.4.3.5 Preference Events

Currently the API specifies a class called `PreferenceChangeEvent` for notifying Xlets of changes to individual preferences. This class only contains information about which Preference has been affected, but not what was done with it. In the NIST RI we have implemented the `PreferenceChangeEvent` class so that it can carry additional information about what was done to the individual Preference. Currently this is only done with respect to `FavoriteChannels` preferences, with two events defined: Channel Added and Channel Removed.

10.4.3.6 Favorite Channels

10.4.3.7 Rating Preference

10.4.3.8 Preferred Language

10.4.3.9 Personal Data

10.5 Application (Xlet) Implementation

10.5.1 Packages javax.tv.xlet and org.atsc.application

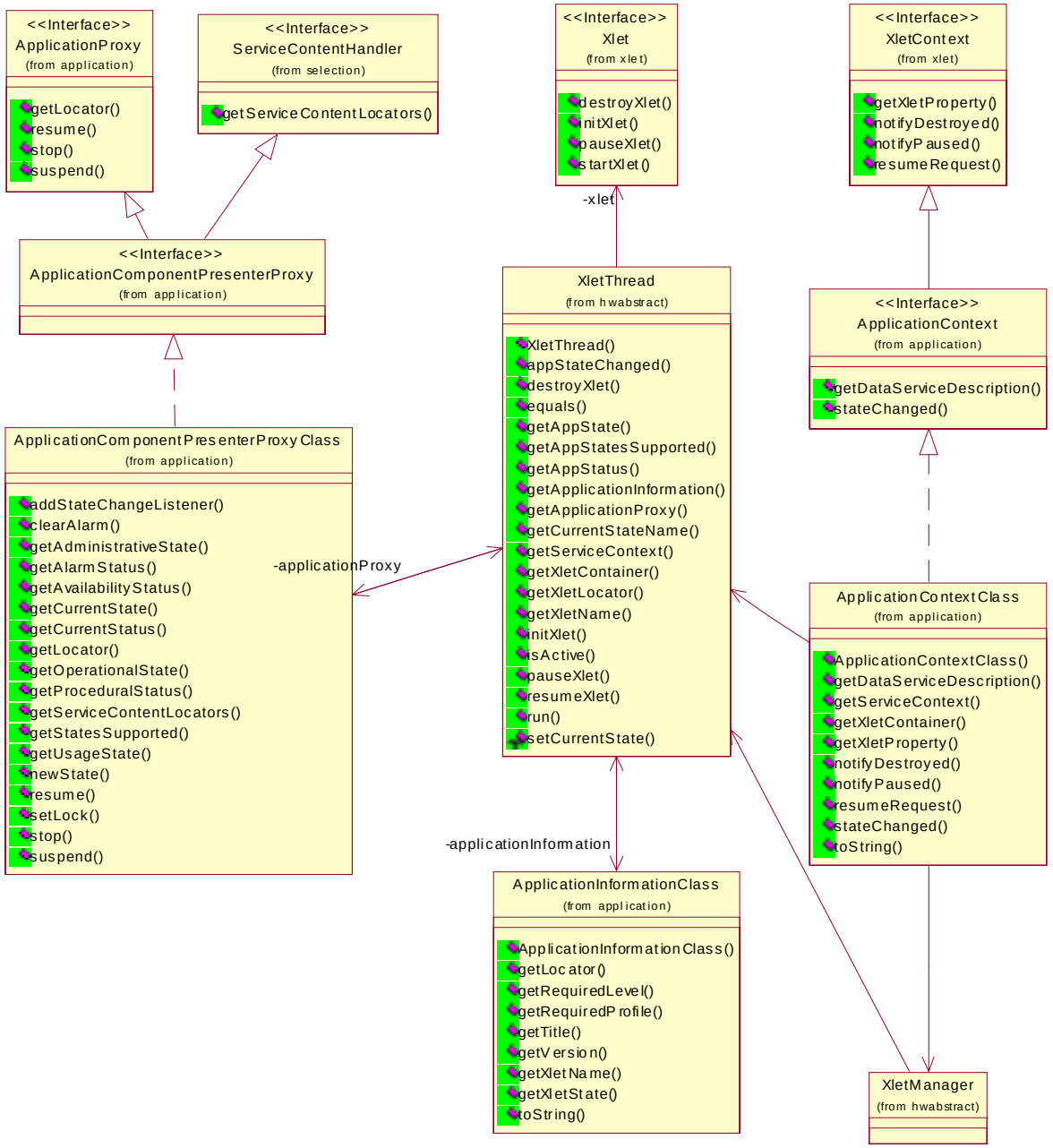


Figure 28 Xlet/Application Implementation

Specification	Implementation Mapping	Level
Interfaces		
Xlet	Implemented by the Xlet. Supported by gov.nist.hwabstract.XletManager, and gov.nist.hwabstract.XletThread	

XletContext	gov.nist.application.ApplicationContextClass	
ApplicationComponentPresenterProxy	gov.nist.application. ApplicationComponentPresenterProxyClass	
ApplicationProxy	gov.nist.application. ApplicationComponentPresenterProxyClass	
ApplicationContext	gov.nist.application.ApplicationContextClass	
ApplicationInformation	gov.nist.application.ApplicationInformationClass	
Classes		
XletStateChangeException	javax.tv.xlet.XletStateChangedException	
Application*Exception	org.atsc.application.Application*Exception	
Application*Cause	org.atsc.application.Application*Cause	

Table 13 Xlet/Application Implementation Mapping

In the DASE view, an Xlet is a DASE Application. Package `org.atsc.application` provides DASE-specific extensions to the Xlet mechanism. For example, interface `org.atsc.ApplicationContext` extends the JavaTV interface `javax.tv.xlet.XletContentext` in order to allow for DASE state information to be applied to a Xlet. Figure 28 shows the class diagram for Xlet and DASE Application interfaces and the associated implementation classes. Management of Xlets is discussed in Section 9.6 where control of the Xlet via the `XletThread` object is described.

The class diagram shows the relationships between the DASE interfaces and the implementation classes. The implementation classes `ApplicationComponentPresenterProxyClass` and `ApplicationContextClass` rely on the `XletThread` class to maintain information associated with the Xlet, such as the Xlet's state and Locator. The `XletThread` class was described in Section 9.6.2.2. An object of this class is executed in it's own thread and calls the Xlet's methods `initXlet(), startXlet(), pauseXlet(), and destroyXlet()`. The Xlet can communicate to the Application registry via the `XletContext` object, which in a DASE system is an `ApplicationContext` object. The implementation class, `ApplicationContextClass` then communicates the request back to the Application Registry object, which in the DASE implementation is the `XletManager`.

When the `XletManager` loads an Xlet, an `XletThread` object is created. The constructor of this class creates the `ApplicationComponententPresenterProxyClass`, `ApplicationInformationClass` and `ApplicationContextClass` objects. Therefore, there are single instances of these objects associated with the Xlet. These classes implement their respective DASE interfaces as can be seen in the class diagram. The `XletThread` class also creates the `XletLocator` object.

Need text and diagrams here to describe how Xlets are managed within the Service/Data Service contexts.

10.6 Data Broadcast API

10.6.1 Introduction

The `org.atsc.data` package provides APIs to access the Service Definition Framework (SDF) of the ATSC T3/S13 Data Broadcast protocol (A/90). Two major components of the SDF are the Data Service

Table (DST) and the Data Event Table (DET). The purpose of the DST is to identify and describe the components of a data service. The DST gives the type, location, and intended use of the data service. It is important to note that the current ATSC DASE specification limits the number of concurrent data services per virtual channel to one. The purpose of the DET is to announce data services. The APIs in this package focus on obtaining a handle to the data service and retrieving electronic program guide information about the data service.

10.6.2 Background

A data service is a collection of applications delivered in the DST. Data service applications can be procedural (Java Xlet) or declarative (XHTML page) applications. Associated with the data service applications are resources. Resources can be files (data) and streaming data (*as currently defined in the ARM, later this will be expanded, see table XXX in A/90). The data files are accessed via data carousels (see section [data carousel]. Figure 29 shows the relationship between virtual channels, data services, data service applications, and resources.

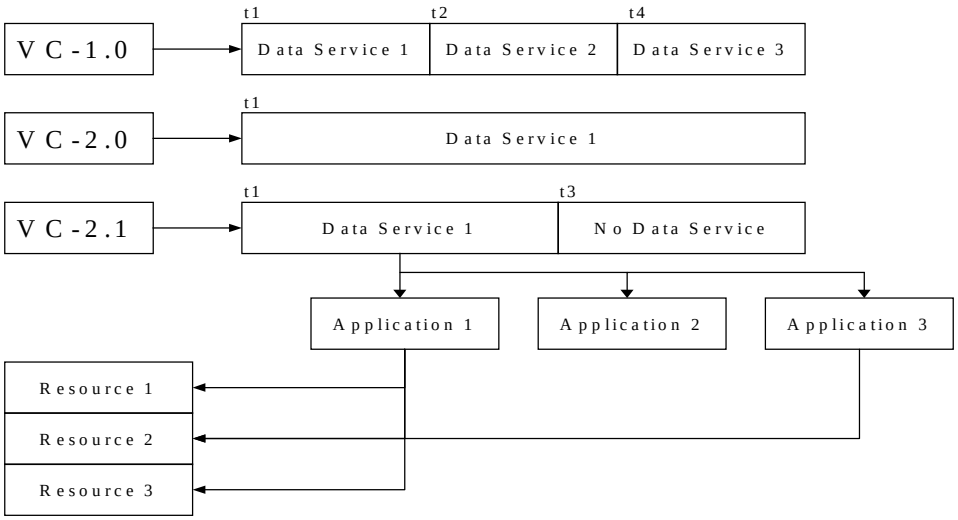


Figure 29 Data Service Overview

The resources associated with a data service application can be files, XXX, or XXX. Resources are accessed in the DST by tap loops. A tap id is used by receiver software to reference a resource. A resource location can be local or remote. A local resource indicates that it is in the current VCT. A remote resource may be a resource on the Internet, on the same transport stream but different VC, or on a different VC in a different transport stream.

The Data Broadcast APIs serves two main functions: access to the data service and announcement of the data services. Section 10.6.3 covers the announcement of data services and section 10.6.6 focuses on obtaining information about the data services.

10.6.3 Data Service Announcement

Data service announcement allows an Xlet to discover EPG type information about the data services available in Service objects. This information can be contained in either the EIT or DET. If the data service is a stand-alone data service, the announcement is made in the DET. If the data service is a data-enhanced A/V channel, then there are optional ways to announce the data service. One option is to jointly announce the A/V and data service in the EIT. A second option is to announce the A/V in the EIT and

separately announce the data service in the DET. Figure X depicts the various scenarios for the announcement of data services. Given these options the implementation needs to take this into account.

An Xlet can obtain data service announcement information in the following way. The Service object obtains a Data Service Details object, which contains a method to access a Data Schedule for the Service. The Data Schedule contains methods to obtain information about the current and future data events. The Data Event object contains the elementary data, such as the start and end time of the data service. Figure X shows the UML diagram for the important interfaces and classes for the data service announcement APIs.

As mentioned the elementary component in data service announcement is the Data Event. Data Events are supported in the HAL layer by two classes, namely the DataEventTable and the HALDataEvent. The DataEventTable is the conglomorence of all DETs for all VCs in the broadcast streams. This class contains methods to access the entire table or to select HALDataEvents for a given VC, or an individual HALDataEvent. The HALDataEvent represents an instance of the ATSC PSIP defined DET. Access to the HAL DataEventTable is via source and data ids. Table 14 depicts where the implementation retrieves information for satisfying a Data Event object.

Method	HAL Object	MPEG/PSIP Table/Attribute
getStartTime()	HALDataEvent	DET/start_time
getEndTime()	derived	None
getDuration()	HALDataEvent	DET/length_in_seconds
getEventName()	HALDataEvent	DET/title_text
retrieveDescription()		ETT
getRating()		
getChannel()		

Table 14 Data Event Implemetation to Transport Mapping

10.6.4 Data Broadcast API Implementation Mappings

Specification	Implementation Mapping	Level
Interfaces		
Compatibility	gov.nist.data.CompatibilityClass	
DataEvent	gov.nist.data.DataEventClass	
DataEventDescription		
DataSchedule	gov.nist.data.DataScheduleClass, RetrieveDataEvent, RetrieveFutureDataEvent, RetrieveFutureDataEvents, RetrieveNextDataEvent, RetrievePresentDataEvent, RetrieveDataServiceDescription	
DataServiceApplication	gov.nist.data.DataServiceApplicationClass	
DataServiceDescription	gov.nist.data.DataServiceDescriptionClass	
DataServiceDetails	gov.nist.data.DataServiceDetailsClass	
Classes		
DataServiceChangeEvent	gov.nist.data.DataServiceChangeEvent	

Table 15 Data Broadcast Implementation Mapping

Notes from Table 15

- In the ATSC DASE specification, the DataServiceDetails interface extends the JavaTv ServiceDetails interface. It may not always be the case that a Service includes Data Broadcast

related information. Therefore, the Xlet may choose to retrieve either of these two ServiceDetail objects. In the NIST implementation, DataServiceDetailsClass extends ServiceDetailsClass (see gov.nist.service.navigation for more details) and implements DataServiceDetails.

- The implementation of DataSchedule relies on a number of asynchronous helper classes, that are used to gather a collection of DataEvents. These classes are identified by the prefix Retrieve. See section X.X on asynchronous data retrieval for more information.

10.6.5 Issues and Notes

Compatibility: The compatibility descriptor may be used to specify data receiver hardware and/or software requirements for proper acquisition and referencing of a data service. The org.atsc.data.Compatibilty interface allows access to this information. The descriptor is present in A/90 Table 12.3. The details of the descriptor are shown in Table 6.1 (DSM-CC Compatibility Descriptor). The interface gives access to the raw content of the descriptor only, the specific fields are not exposed with API method calls. Table 12.3 has a Compatibility Descriptor for the Data Service Application. This is a DSM-CC Compatibility Descriptor in Table 6.1. This table then has a loop of the actual Compatibility Descriptor information.

Questions/Comments:

1. Why doesn't the Compatibility interface expose the fields of the A/90 compatibility descriptor? The API wants just the raw bytes to be returned. Where is this?

Resources for a Data Service Application*:* Not really sure about this. Resources for a Data Service Application are signaled through the entryPoint, classPath, and Decoder subdriptors (which are part of the Application Descriptor). EntryPoint identifies data resources to be auto-launched in order to start the application. ClassPath identifies data resources to go in the ClassPath of a Java Application. Decoder identifies a resource as a special (Java) decoder.

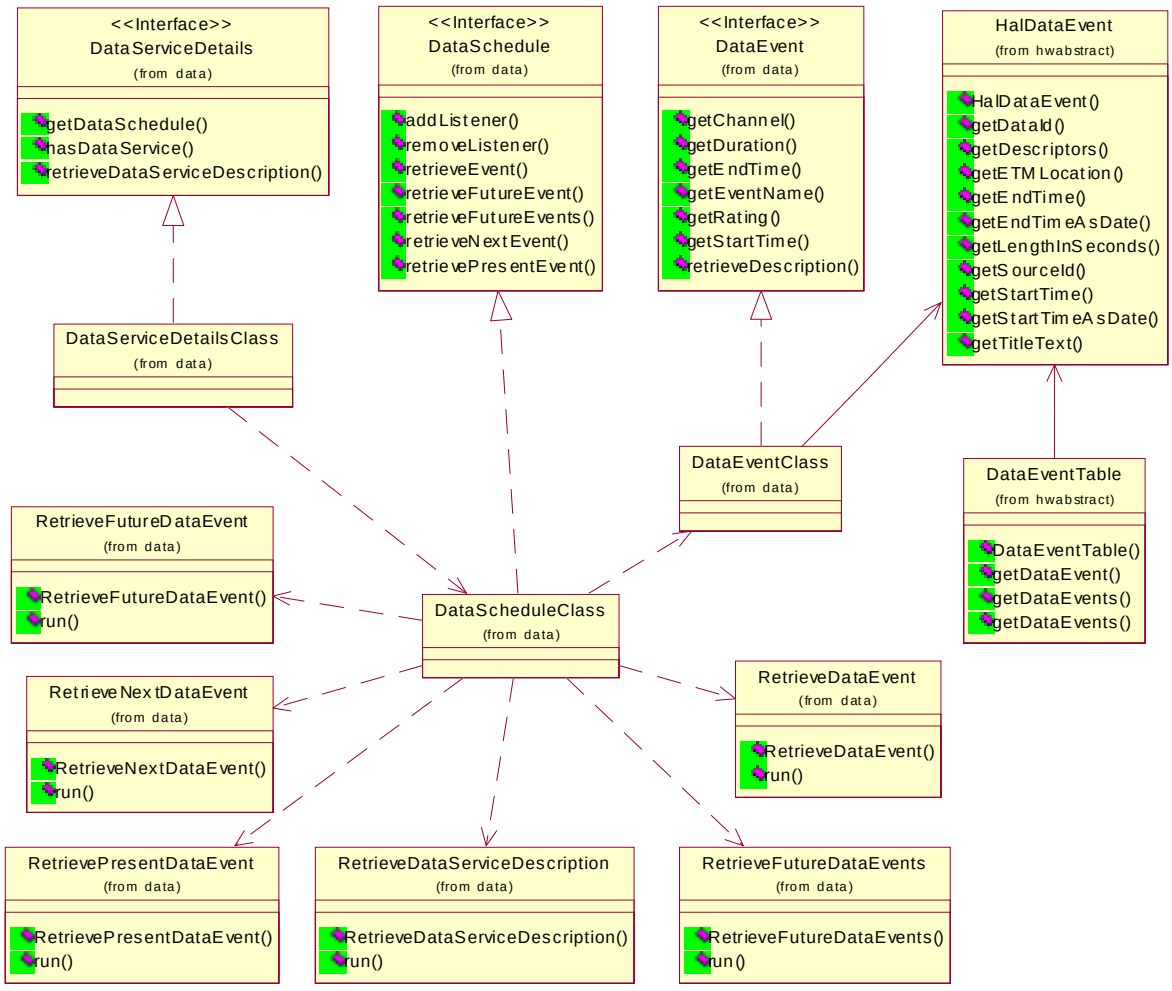


Figure 30 Data Service Announcement

10.6.6 Data Service Access

Data Service access allows an Xlet to gather information about a particular data service. This information is obtained in the DST. From the API point of view the Xlet accesses this information by using its context (org.atsc.application.ApplicationContext) to retrieve a DataServiceDescription object. The DataServiceDescription object abstracts the information of an instance of a DST. The DataServiceApplication object represents an instance of a data service application. Figure 31 shows the UML diagram for the important interfaces and classes for the data service access APIs.

The API implementation relies on HAL classes for identifying data services. The HAL DataServiceList is a collection of all the data services (instances of the DST) across all transport streams. The link to this into this list is made with the program number attribute in the virtual channel.

DataServiceDescription (A/90 DST table 12.2, data_service_table_bytes, 12.3)

Attribute	HAL	Transport

Application		application loop in A/90 Table 12.3
Title		title descriptor
Locator		???
PrivateData		raw private data loop in A/90 Table 12.3

10.6.6.1

10.6.6.1.1 DataServiceApplication

Attribute	HAL	Transport
ApplicationId	DBDaseApplicationContext	12.3/App Loop/app_id_byte (app_id_byte shall be set to a lid: string value to globally and uniquely identify the application). app_id_byte loop contain a number of bytes representing a String URI. Follow appIdUri in DBDaseApplicationContent
Compatibilities		
Title		
PrivateData	DBApplicationContent	12.3/Application Loop/app_data_byte
ResourceLocator		
ResourceNames		

Table 16 Data Service Application Implementation to Transport Mapping

10.6.6.1.2 Accessing the Data Services from the HAL

1. Access the HAL DataManager to retrieve the DataServiceList. The DataServiceList contains a list of all available Data Service Tables in the A/90 sense. For a given Virtual Channel, a Data Service is retrieve by calling the getDataServices() method. The link is made with the programNumber. This method returns an array of Data Services for this Virtual Channel. The first element contains the Data Service. Caution, future versions of the DASE specification may allow more than one Data Service per Virtual Channel.

2. Accessing the title for the Data Service. The title is accessed in the titleDescriptor, which is a DASE specific descriptor defined in the ARM document. The descriptor can be present as a descriptor in the service info descriptor loop in the Data ServiceTable Byte Structure (A/90 Table 12.3). The API implementation accesses this Descriptor data through the DataService and obtains a Descriptor List. This list is search for the present of the title descriptor. **Future: Maybe write a general procedure for HAL descriptors.**

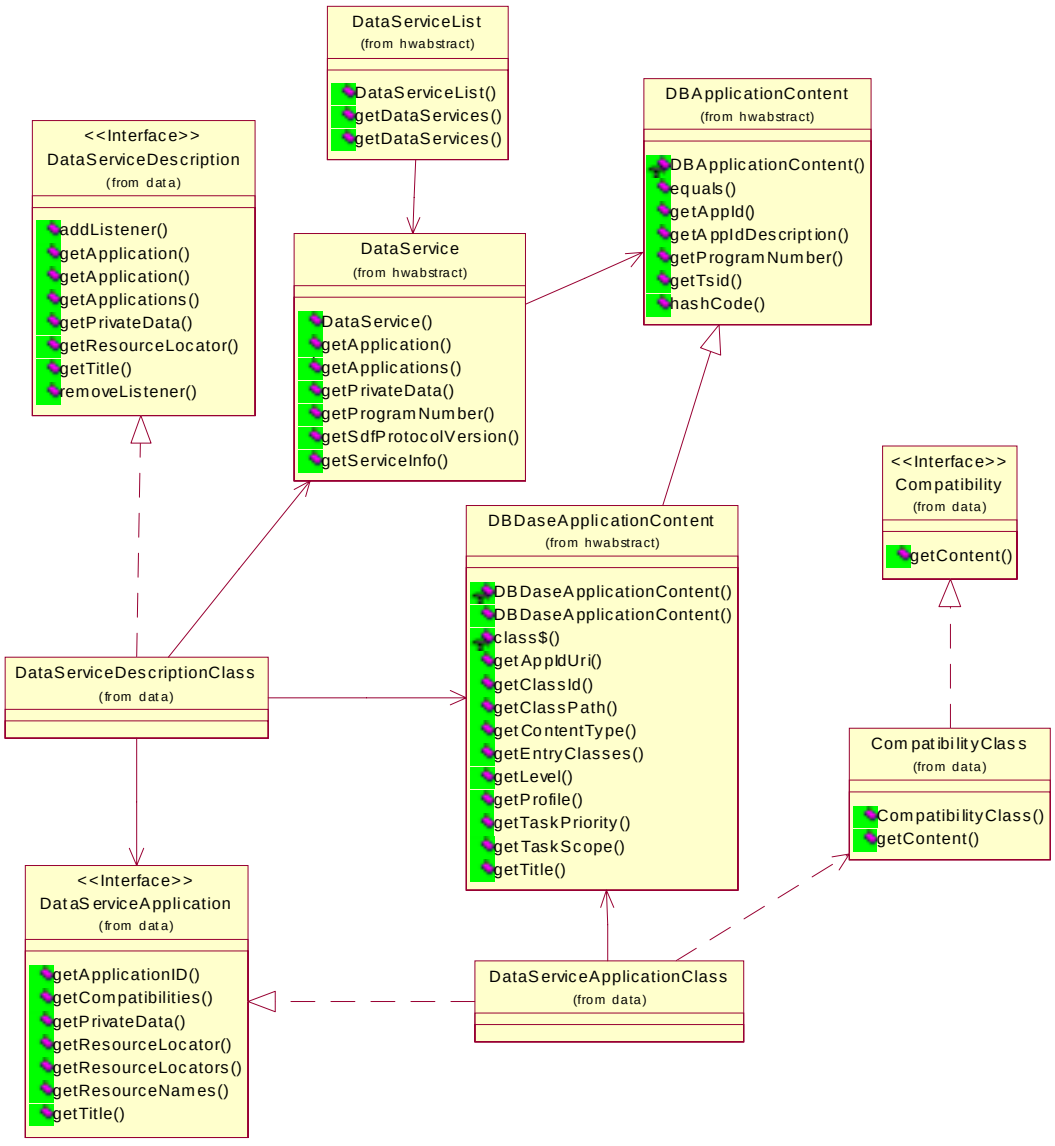


Figure 31 Data Service Access

10.7 System and TV Graphics API

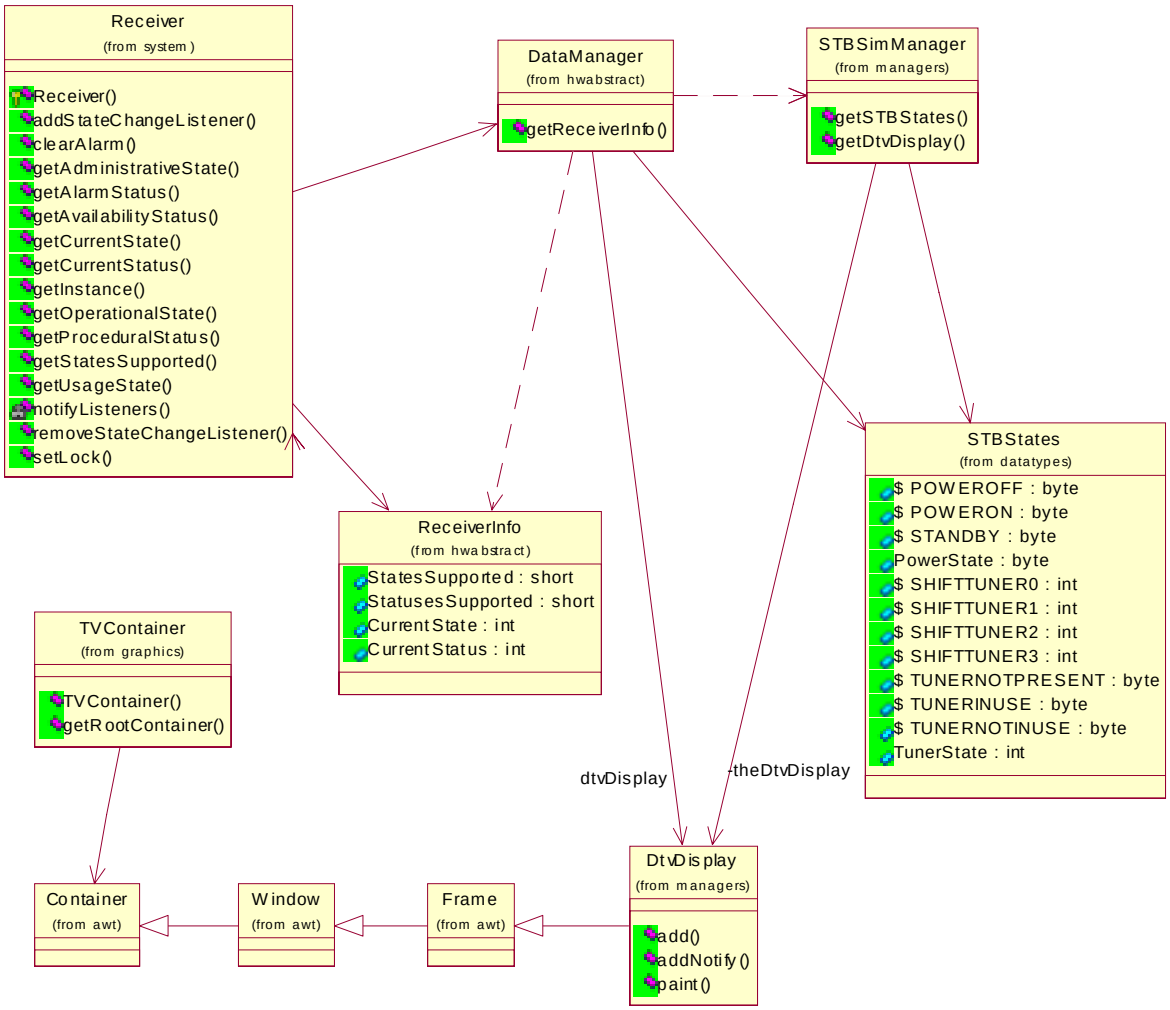


Figure 32 System API Implementation

The `org.atsc.system.Receiver` class is implemented to receive the status of the receiver from the hardware abstraction layer. The HAL then receives the status from the underlying set-top box implementation. In the NIST RI, the simulation provides this information as shown in the class diagram in Figure 32. The HAL `DataManager` class maps the simulation states from the `STBSates` class into states and status needed by the `org.atsc.Receiver` class.

Classes `DtvDisplay` and `STBStates` are part of the STB simulation. The `javax.tv.graphics.TVContainer` class also is implemented to receive its information from the HAL. The STB simulation provides management of a DTV display by allocating an object that implements the `java.awt.Container` interface that is needed by `TVContainer`. The diagram shows that class `DtvDisplay` does implement the `Container` interface by implementing the sub-interface `Frame`.

10.8 The Networking API

10.9 The Registry API

Figure 33 Registry Implementation

Several registries need to be supported within a DASE environment. The details of the various registry implementations are described in the appropriate sections. The Application Registry is described in Section 9.6.2.1. The User Registry is described in Section 10.4.1and the Preference Registry is described in Section 10.4.3.

10.10 The Document Object Model (DOM) API

The Document Object Model (DOM) API is used to allow Xlets to interact with and control declarative portions of an application. This API is specified in packages `org.atsc.dom`, `org.atsc.dom.html`, `org.atsc.dom.legacy` and `org.atsc.dom.views`. The classes and interfaces

in these packages are based on classes and interfaces defined in package `org.w3c.dom` and `org.w3c.dom.html`.

These ATSC classes and interfaces are not currently implemented in the NIST RI. However, files do exist for the interfaces and classes in the source tree, but are not compiled as part of the current build system.

10.11 The Trigger API

The Trigger API is defined in packages `org.atsc.trigger`. This API is not currently implemented in the NIST RI although stub code does exist in the source tree.

10.12 HAVi UI

10.12.1 Current Status

Insert content of README here.

10.12.2 Remote Control

10.12.3 Supported Devices

10.12.4 Looks

10.12.5 Widgets

10.13 DAViC

10.13.1 Introduction

10.14 Complete Data Flow Examples

10.14.1 Introduction

The HAL objects communicate with the simulation engine via the public interface of the `STBSimManager` class. This interface defines several methods used to access the ATSC tables as well as the other data, such as users, preferences, etc. One such method, `getATSCDataManager()` returns an `ATSCDataManager` object which provides access to the ATSC table data.

The class `ATSCDataManager.ATSCTableSetRequest` contains a set of Boolean member variables. The requester sets to true each member variable for which a ATSC table reference is needed. For example, `ATSCDataManager.ATSCTableSetRequest.vctRequested` would be set to true to indicate that the ATSC Virtual Channel Table is requested. The class `ATSCTableSet` defines a set of public

member variables that are arrays of the individual ATSC table elements. For example, `ATSCTableSet.vct` is a reference to the current ATSC virtual channel table. A *NULL* reference indicates that the table is not currently available, or wasn't requested. It is up to the requestor to reconcile what tables were requested against what references are returned.

Most of the ATSC tables are treated as a synchronized set by the `ATSCDataManager`. The System Time Table (STT) is one exception, because it is independent of the other ATSC tables. The Master Guide Table (MGT), Virtual Channel Table (VCT), Event Information Table (EIT), Extended Text Table (ETT) and Rating Region Table (RRT) are all updated as a complete set. Therefore, when the HAL requests table data, all of the listed tables will be synchronized by the call to `ATSCDataManager.getATSCTables()`. In other words, the tables returned all belong to the same set. Future calls to `getATSCTables()` may return newer versions of the tables than a previous call.

The simulation handles multiple input streams by providing multiple sets of tables to the hardware abstraction. The tables are not merged into a single database. The hardware abstraction is responsible for merging all of the data into a single view based on the needs of the DASE API.

Notification of changes to the data is accomplished by using the STB Change events. The client of the STB simulation can register with the `STBSimManager.addSTBChangeListener()`method, passing in an object implementing the `STBChangeListener` interface. When changes to the data are made by the table managers, the `STBSimManager` object will notify all registered listeners by calling method `STBChangeListener.STBChange()` with an `STBChangeEvent` object. Method `STBChangeEvent.getSTBChangeInfo()` returns a `STBChangeInfo` object which indicates which data items have changed.

The HAL objects obtain a reference to the simulation manager object by calling a native function. This call is needed because native process `STB_main` creates the simulation manager object. After obtaining a reference to the simulation manager, all future access to the simulation database is done via the simulation manager. The reason for this is to maintain the synchroneity between all of the ATSC tables.

When the DASE application is executed, it first creates a factory object. The purpose of the Factory is to isolate the application from the specifics of the API implementation (such as class names). A DASE application uses the factory object as a starting point for communication with the DASE API. Factory classes are specified in the DASE API document.

The factory class returns an object in the API (`SIManager` for example). This object obtains a reference to the HAL data manager by calling method `getHwManager()`in class `gov.nist.hwabstract.STBEnvironment`. This object encapsulates the access to the existing data manager object and enhances portability of the API.

10.14.2 Service Information Example

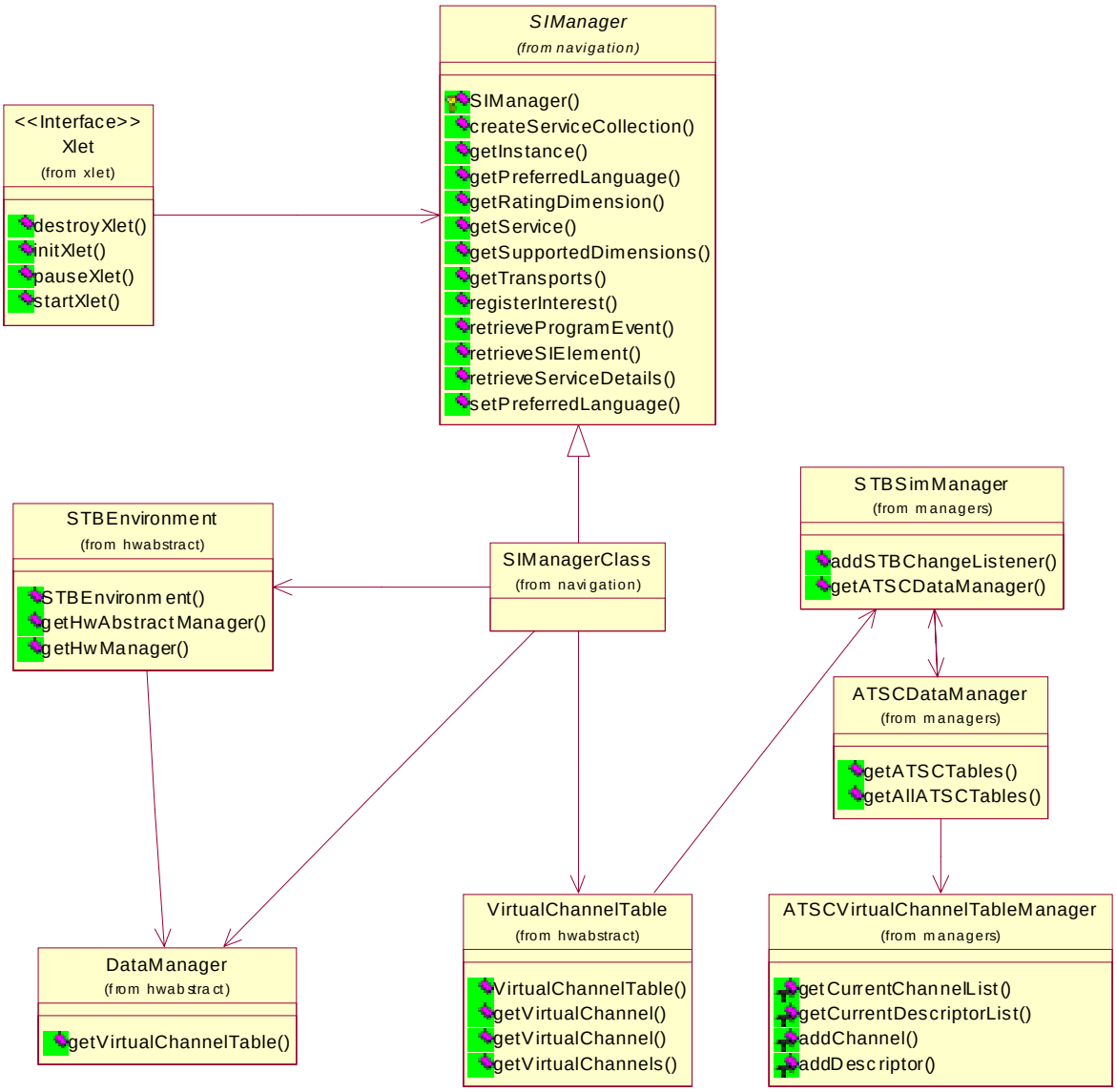


Figure 34 Class Diagram for Service Information Retrieval

Figure 34 shows the UML class diagram for the System Information (SI) Manager to Simulation interaction. This diagram illustrates the management of the ATSC virtual channel data, and the retrieval of the data by the DASE application.

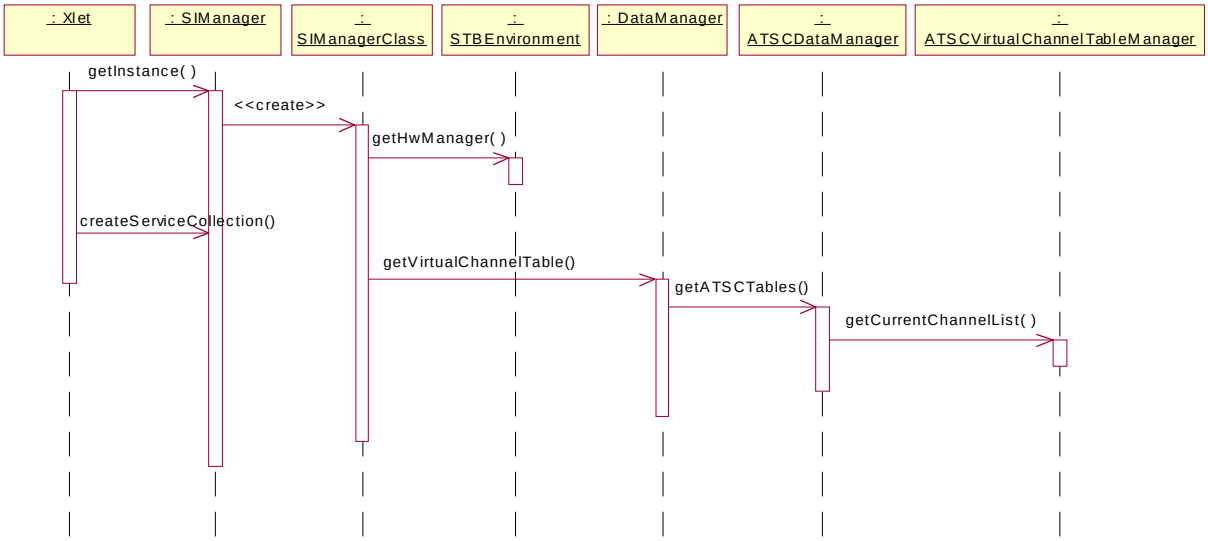


Figure 35 DASE Service Retrieval

The DASE application first calls static method `SIManager.getInstance()` that returns a reference to an object of class `SIManagerClass`. This reference is a `SIManager`, the abstract super class of `SIManagerClass`. `SIManager` is defined in the DASE API, and `SIManagerClass` implements the `SIManager` methods.

The `SIManagerClass` object makes a call to `STBEnvironment.getHwManager` to obtain a reference to the HAL data manager. The constructor of `STBEnvironment` calls native method `getHwAbstractManager()` (not shown in the diagram) to retrieve a reference to the hardware abstraction data manager from `STB_main` which created the data manager object.

At this point, the DASE application can call the methods of `SIManager` to retrieve information. In Figure 35 the application (Xlet) is calling `SIManager.createServiceCollection()`to obtain a collection of Services.

The SI manager then calls the HAL `DataManager.getVirtualChannelTable()` to retrieve the Virtual Channel table. The `DataManager` calls simulation object `ATSCDataManager.getATSCTables()` to retrieve the current versions of the data tables. This call results in a call to `ATSCVirtualChannelTableManager.getCurrentChannelList()` to retrieve the virtual channels.

The `SIManagerClass` object registers with the `STBSimManager` as an `STBChangeListener` in order to receive notification of updates to the ATSC tables. When a new set of tables arrives, the new table data is copied in to the respective manager in a synchronized manner and an `STBChangeInfo` object is sent to all registered listeners.

11. SECURITY

12. JAVA RUNTIME ENVIRONMENT EXTENSIONS

The Java Runtime Environment (JRE) extensions implement added functionality to several `java.io` classes. However, because changes were made to some Java source code that is part of the Sun Java Development Kit, these changed files are not delivered as part of the NIST RI. This appendix will describe the changes needed, however.

The following classes have been modified in order to provide the capability to read from Carousel files:

`java.io.FileReader`

`java.io.FileInputStream`

`java.io.RandomAccessFile`

A new class, `CarouselFileConnection` has been added to the `java.io` package. This is a static class that manages a database of `FileDescriptors` for Carousel Files. It really is the Carousel File System: all opened Carousel Files are registered in it. Finally, a new 'C' library (with filename `jreX`) was created to access the native implementation of some `java.io` functions.

The general idea was to implement in Java a branched treatment of `java.io` functions to replace the previously native implementation. In the case of a regular file, we call the JRE native implementation through the `jreX` library; in the case of a carousel file, the code is present in the function itself. Here is an example for the `read()` function of `java.io.FileInputStream`:

The `read()` method was replaced with:

```
public int read() {
  if(fileIsCarouselFile) {
    // FileInputStream instantiated
    // around a carousel file.
    // Java implementation
  } else  {
    // FileInputStream instantiated
    // around a regular file
    // native implementation
    return readBridge();
  }
}
/* This new native function allows
* to go around the name conflict
* and access old native java.io implementation.
* readBridge is implemented by jreX.c.
*/
public native int readBridge();
```

With the corresponding implementation in `jreX.c`:

```
/*
```

```
           * Class:     java_io_FileInputStream
           * Method:    readBridge
           * Signature: ()I
           */
          JNIEXPORT jint JNICALL Java_java_io_FileInputStream_readBridge
                (JNIEnv * env, jobject thisObj) {
return Java_java_io_FileInputStream_read(env, thisObj);
```

The same scheme was applied to all functions.

13. APPLICATIONS

This may become a separate document..

13.1 A Prototypical Xlet

This section shows an example Xlet where the minimal functionality is implemented. Such functionality includes implementing the Xlet interface, notifying the Application Registry of state changes via the `ApplicationContext` object, and cleanly exiting the runtime environment.

To be continued….

13.2 *The Electronic Program Guide Xlet*

One of the targeted applications for the DASE environment is a downloadable Electronic Program Guide (EPG). An EPG provides viewer information about current and future programming and resembles the TV Guide page in a newspaper (Figure 36). This information can be displayed as a simple overview of the programming or as detailed descriptions. An EPG uses data from the Service Information (SI) database. The SI database is a collection of tables describing current and future programming and is made up of PSIP tables described earlier. The Service package within the DASE API contains functionality for service (channel) navigation and selection.

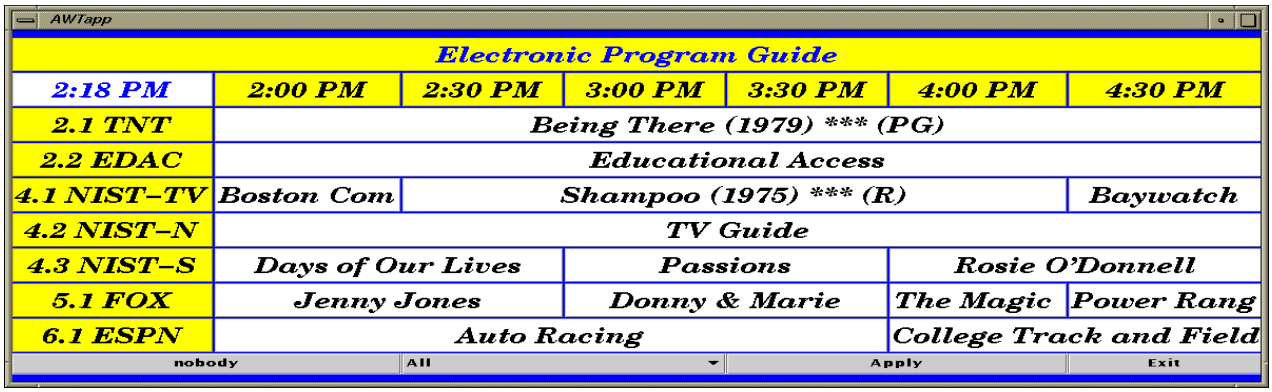


Figure 36 Electronic Program Guide

Figure 37 gives a sequence interaction diagram for an EPG. It gives an overview of how an application would use the API to obtain SI database information for a program guide. The DASE application first gets an instance of the *SIManager*. The *SIManager* is the access point into the SI database. Once access to the SI database is established, one approach to building an EPG is to get a list of all available *Services*. This can be accomplished with the createServiceCollection() method. If called with no filtering criterion, this method returns a list of all known *Services*. The list can be sorted and *Services* can be retrieved in order. Next the DASE application would extract the Name (e.g., PBS) and the *ServiceNumber* (i.e., the channel number) for each *Service* in the *ServiceCollection*. This data is shown in the left-most column of the example EPG illustrated in Figure 36. Next the application needs to obtain the program events for each *Service*. This is accomplished by getting a *ServiceDetails* objects for every Service by using the retrieveDetails() method. The *ServiceDetails* object contains a method to extract the *ProgramSchedule* for the *Service*. The *ProgramSchedule* contains the list of *ProgramEvents*. The retrieveCurrentEvent() method can be used to get the *ProgramEvent* that is currently showing on that service. *ProgramEvent* information includes the name of the program, the starting and ending times, and the program rating. An extended

description of the *ProgramEvent* details can also be retrieved. This process of retrieving *ProgramEvents* would be continued with the method retrieveNextEvent(). Also, the *ServiceCollection* would be looped through for subsequent *Services* by using the getNext() method call of *ServiceCollection*.

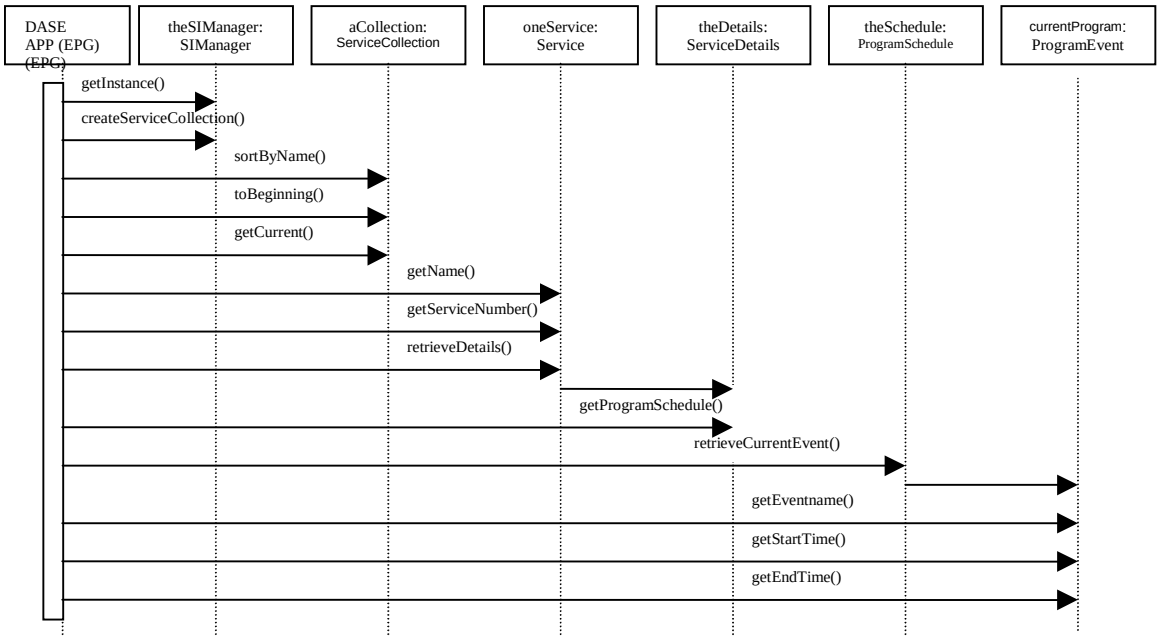
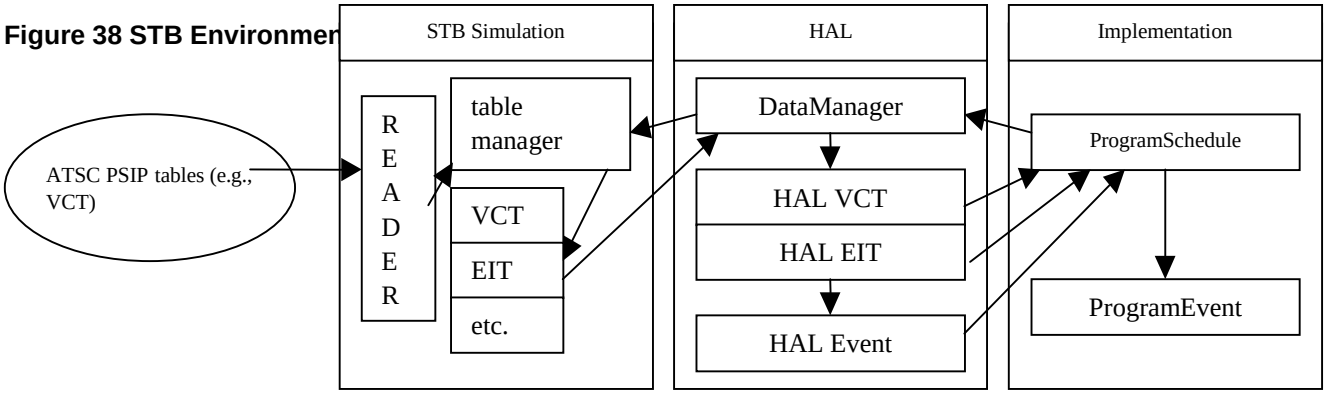


Figure 37 EPG Sequence Interaction Diagram

Figure 38 illustrates how the objects at the API implementation level obtain data from the STB environment. The example shows a slice of the data gathering process when retrieving *ProgramEvents*. Each *ProgramSchedule* object has an associated HAL virtual channel. The *ProgramSchedule* obtains a reference to the HAL *DataManager* and uses it to get a reference to the HAL EIT. A list of HAL Events is then obtained by gathering all events for the associated virtual channel. These events are used to create API level *ProgramEvents*. The current event is found by using the time of day on the set-top box to search the HAL event list. Once the event is found a *ProgramEvent* object is built by the API.

The HAL EIT obtains its data from the STB simulation ATSC EIT manager. The representation of the EIT at the simulation level is a direct mapping of the ATSC PSIP table. The simulation receives the table information that is extracted from the DTV bitstream and creates Java objects that are returned to the HAL. The HAL EIT is responsible for merging data from the various simulation tables.

Figure 38 STB Environmer



13.3 The Stock Sticker Xlet

13.3.1 Introduction

The Stock Ticker Xlet presents dynamic stock quote information to a television screen in a DASE environment (see Figure 39). The Xlet uses the Data Carousel APIs to obtain the broadcast data. The data file is sent as data carousel modules with an associated universal resource identifier (URI). The data contains the stock symbol, current price, delta, and time of the quote. The Xlet reads this information via the data carousel API and displays the information to the television screen. The StockTicker Xlet demonstrates the use of the Data Carousel, Xlet, and Locator APIs.

Symbol	Price	Change
GE	47.54	+0.24
WMT	55.75	+1.50
AMD	26.25	-2.50
TXN	39.00	-1.25
EBAY	46.50	+0.75

Figure 39 Stock Ticker Display

13.3.2 Components

The Stock Ticker Xlet depends on a number of Java class file components. StockTicker and StockDataModel are the Xlet components. StockStreamer is the data source module. QuoteList is the bridge module, where StockStreamer writes data, and StockTicker reads it. QuoteFileGen is a utility tool to create the initial set of stock quotes. Below a summery of each component is given. The interactions between the components are given in Figure 40. Note that the StockTicker, StockDataModel, and QuoteList comprise the Xlet components. StockStreamer and QuoteFileGen are utility components in the DASE environment that are use to stream stock quote and to build stock quote data.

StockTicker: This is the Xlet class that displays the stock ticker. It is responsible for opening and reading from the carousel file. After successfully reading stock quote data it displays the stock ticker to the television screen. StockTicker uses the StockDataModel to help create the stock ticker table. The StockTicker Xlet implements the Xlet and CarouselFileListener interfaces. It receives quote updates via the CarouselFileListener.

StockDataModel: This class provides the AbstractTableModel for the JTable swing component (note that HAVi components will replace the swing components in future version of the StockTicker). At this point we assume that the swing class files are broadcast as part of the StockTicker application.

StockStreamer: This class is responsible from generating a stream of stock quotes. It creates a stream of QuoteList objects and writes these to the data carousel (FIFO). This is a stand-alone component.

QuoteList: This class is an instance of the stock quote data. QuoteList contains the entire list of quotes. QuoteList has an inner class (Quote) that represents a single stock quote. A Quote consists of a Symbol (stock name), a Price (current value), a Delta (change since the last closing price), and the time of the quote.

QuoteFileGen: This class generates the initial set of stock symbols. The StockStreamer reads the created file.

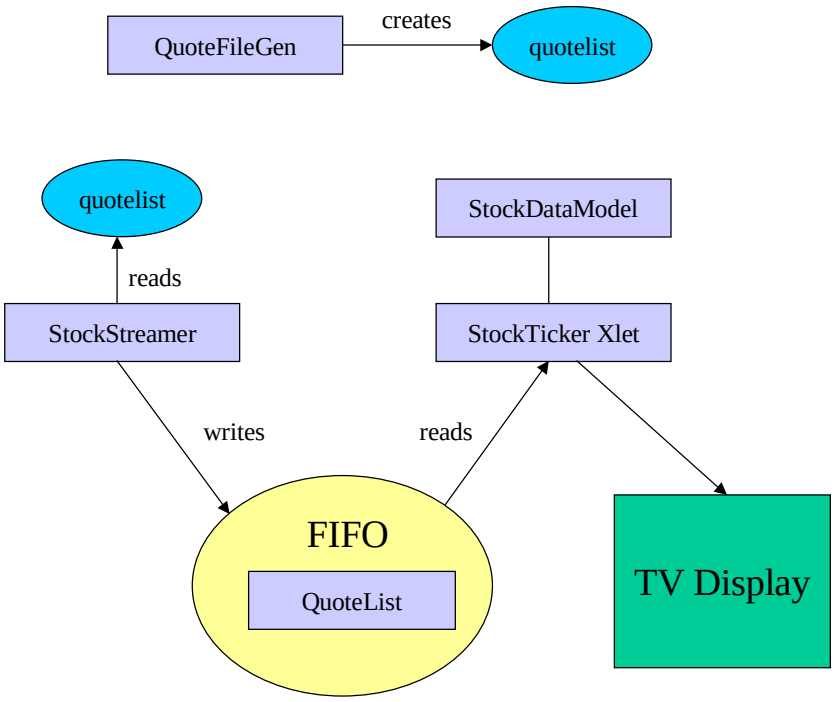


Figure 40 StockTicker Components

13.3.3 The StockTicker Xlet

The major tasks of the Stock Ticker are given below.

- Establish a connection to the carousel file.
  - Get a LocatorFactory instance
  - Create a Locator from the URI
  - Transform Locator
  - Create CarouselFile object
- Read the stock quote data
  - Create a FileInputStream object using the CarouselFile
  - Create an ObjectInputStream using the FileInputStream
  - Read a QuoteList object from ObjectInputStream
  - Obtain an array of Quotes from the QuoteList object
- Build the Stock Ticker table
  - Initialize the StockDataModel
  - Create the Stock Ticker table.
  - Bind the Stock Ticker table to the TVContainer.

- Update the Stock Ticker table.
  - Add Xlet to list of data carousel listeners
  - Refresh data carousel cache upon receipt of data carousel update
  - Read a QuoteList object from ObjectInputStream
  - Update StockDataModel with new Quote data

13.3.4 Compiling (Unix)

This Xlet requires that the jreX extensions for the data carousel be compiled in. Do this by including the --with-jrex flag during configure. In the nist_ri root level directory, use the following commands. Make sure that the distribution you have includes the jreX extensions.

```
% make distclean
% configure --with-jrex
% make
```

13.3.5 Setting up the Run Environment (Unix)

1. Enable the Data Carousel Simulation Runtime Extensions

To enable the data carousel runtime extensions the jreX.jar file must be added to the STBSIM_CLASSPATH and must be the first file. In addition, the standard Java runtime jar file, rt.jar needs to be the second file in the STBSIM_CLASSPATH environment variable. A sample setenv STBSIM_CLASSPATH command is given below.

```
% setenv STBSIM_CLASSPATH \
${HOME}/nist_ri/simulation/runtime/jreX.jar: \
/usr/java/jre/lib/rt.jar: \
${HOME}/nist_ri/simulation/runtime/stb.jar: \
${HOME}/nist_ri/simulation/runtime/dase.jar: \
${HOME}/nist_ri/simulation/runtime/hwabstract.jar: \
${HOME}/nist_ri/simulation/runtime/devkit.jar: \
/usr/java/jre/lib/jmf.jar
```

2. Enable the Data Carousel Dynamic Library Path Runtime Extensions

To enable the JRE extensions the STBSIM_LIBRARY environment variable must also be set to contain the location of the libjreX.so library file and the location of the standard Java runtime libraries. Below is an example of how to set the STBSIM_LIBRARYPATH.

```
      % setenv STBSIM_LIBRARYPATH \
${HOME}/nist_ri/simulation/runtime:/usr/java/jre/lib/i386
```

13.3.6 Running (Unix)

Note: all cd commands are relative from the NIST Prototype Implementation root directory (nist_ri).

1. Create quote list file

```
% java -classpath $CLASSPATH applications.xlets.stock.QuoteFileGen GE 45.32
+0.50 WMT 55.75 +1.50 AMD 26.65 -2.25 TXN 39.00 -1.25 EBAY 46.34 +0.75
```

Note: This will create a file named quotelist. Use this file as input to StockStreamer.

2. Start DtvSimulator and start RunXlet utility

```
% cd bin
% ./STB_main -s -x -f /tmp/stockticker.fifo
➢  tools/simulation/RunXlet
```

Note: Go to step 3, then come back to the DtvSimulator program in step 4.

3. Start StockStreamer (in another xterm)

```
% java -classpath $CLASSPATH applications.xlets.stock.StockStreamer  -i
quotelist -o /tmp/stockticker.fifo -pat -t 2000
```

Notes: Make sure the CLASSPATH includes the devkit.jar and stb.jar files. The "-i quotelist" indicates the file to read the initial quotes from. Make sure quotelist is accessible to the StockStreamer application. "-pat" indicates to send in a psip program association table. This table will signify a lid, in this case, lid://nist.gov/data/stockdata. The StockTicker Xlet will use this lid to obtain a locator. The "-t 2000" indicates an interval (in milliseconds) to send the stock quotes. In this case, it is 2 seconds. "-o /tmp/stockticker.fifo" indicates the fifo set with DtvSimulator.

4. Start StockTicker Xlet (under the DtvSimulator, RunXlet program)

```
RunXlet> applications/xlets/stock/StockTicker
```

Notes: RunXlet tells the Application Manager to load and run the Xlet. Make sure the Xlet's class files are located in STBSIM_CLASSPATH. Setting the STBSIM_CLASSPATH as specified above will be sufficient since it contains devkit.jar.

5. Monitoring and Controlling the StockTicker Xlet (not a required step)

The StockTicker can be paused/restarted/destroyed through the use of the Application Control Panel. On the simulated remote control, enable the remote control by pressing the "on" button. Then select the "apps" to start the Application Control Panel. The StockTicker Xlet (or any Xlet) can be control and monitored with this utility.

13.3.7 StockTicker Xlet Source Code (selected modules)

References

[ATSC:API] ATSC T3/S17 (DASE) API Specification

[ATSC:A65] Advanced Television Systems Committee,
Program and System Information Protocol for Terrestrial Broadcast and Cable, Document A/65

[ATSC:A90] Advanced Television Systems Committee,
ATSC Data Broadcast Standard, Document A/90

[UML] G. Booch, J. Rumbaugh and I. Jackson,
The Unified Modeling Language User Guide, 1999 Addison Wesley

[USER-GUIDE] *NIST DASE Development Environment User's Guide*

14. DISCLAIMER

NOTICE OF SOFTWARE ACKNOWLEDGMENT AND REDISTRIBUTION

The software (named NDRI, for NIST/DASE API Reference Implementation) described herein is released by the National Institute of Standards and Technology (NIST), an agency of the U.S. Department of Commerce, Gaithersburg MD 20899, USA. The software presented here is intended to be utilized for research purposes only and bear no warranty, either express or implied. NIST does not assume legal liability nor responsibility for a User's use of a NIST-derived software product or the results of such use.

Please note that within the United States, copyright protection, under Section 105 of the United States Code, Title 17, is not available for any work of the United States Government and/or for any works created by United States Government employees. User acknowledges that this software contains work that was created by NIST employees and is therefore in the public domain and is not subject to copyright. The User may use, distribute, or incorporate this code or any part of it provided the User acknowledges this via an explicit acknowledgment of NIST-related contributions to the User's work. User also agrees to acknowledge, via an explicit acknowledgment, that User has made modifications or alterations to this software before redistribution.